THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Université Toulouse III - Paul Sabatier*
Discipline ou spécialité : *Informatique*

Présentée et soutenue par
Roman Bourgade

Le *22 octobre 2012*

Titre :
Analyse du temps d'exécution pire-cas de tâches temps-réel exécutées sur une architecture multi-cœurs

---

JURY

Mme Isabelle Puaut – Professeur, Université de Rennes I, Rapporteur
M. Bernard Goossens – Professeur, Université de Perpignan Via Domitia, Rapporteur
M. Christian Percebois – Professeur, Université Toulouse III, Examinateur
Mme Christine Rochange – Maître de Conférences, Université Toulouse III, Président

---

Ecole doctorale : *Mathématiques, Informatique et Télécommunications de Toulouse (ED 475)*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse (UMR 5505)*
Directeur de Thèse : *M. Pascal Sainrat - Professeur, Université Toulouse III*
Co-directeur : *Mme Christine Rochange - Maître de Conférences, Université Toulouse III*

Table des matières

Remerciements **7**

Introduction **9**
Contributions 10
Organisation du document 10

1 Analyse de temps d'exécution pire-cas pour systèmes temps-réel strict **11**
1.1 Estimation du temps d'exécution pire-cas 12
1.1.1 Définition de l'estimation de temps d'exécution pire-cas 12
1.1.2 Variabilité des temps d'exécution 12
1.1.3 Approches et méthodes utilisées 13
1.2 Approches dynamiques pour le calcul de temps d'exécution pire-cas 13
1.2.1 Contexte 13
1.2.2 Stratégies d'obtention du pire scénario d'exécution 14
1.3 Analyse statique de temps d'exécution pire-cas 15
1.3.1 Analyse de haut niveau 15
1.3.2 Analyse de bas niveau 20
Conclusion 24

2 Arbitrage, prévisibilité et calcul de temps d'exécution pire-cas pour architectures multi-cœurs **25**
2.1 Vue d'ensemble des différents types d'architectures multi-cœurs 27
2.1.1 Architectures multi-cœurs pour plateformes non mobiles 27
2.1.2 Architectures multi-cœurs pour systèmes embarqués 28
2.2 Partage de ressources dans les architectures multi-cœurs 28
2.2.1 Ressources de stockage 29
2.2.2 Mécanismes d'interconnexion 31
2.3 Prévisibilité du comportement pire-cas des mécanismes d'interconnexion 36
2.3.1 Protocoles et mécanismes d'arbitrage de bus non prévisibles 37
2.3.2 Mécanismes d'arbitrage de bus compatibles avec l'estimation de temps d'exécution pire-cas 40
Conclusion 42

3 Arbitrage de bus multi-niveaux pour des charges de travail hétérogènes 45

3.1 Vers un modèle d'arbitrage multi-niveaux prévisible . . . 46
3.1.1 Un arbitrage de bus à deux niveaux . . . 46
3.1.2 Deux politiques d'ordonnancement des accès . . . 47
3.2 Le protocole *Group Round Robin* . . . 48
3.3 Le protocole *Geometric Group Latencies* . . . 49
3.3.1 Présentation de l'algorithme *Geometric Latencies* . . . 49
3.3.2 Spécification . . . 50
3.3.3 Preuve de la spécification . . . 51
3.3.4 Implémentation matérielle . . . 52
3.3.5 Généralisation à un protocole à deux niveaux . . . 52
3.4 Modalités d'évaluation du mécanisme . . . 54
3.4.1 Méthodologie . . . 54
3.4.2 Caractéristiques de l'architecture cible . . . 55
3.4.3 Stratégies d'allocation des tâches aux groupes . . . 56
3.4.4 Modalités de configuration de l'arbitre . . . 59
3.5 Résultats expérimentaux . . . 61
3.5.1 Test exhaustif de toutes les combinaisons . . . 62
3.5.2 Validation des stratégies optimales d'allocation . . . 67
Conclusion . . . 70

4 Répartition et ordonnancement de charges de travail hétérogènes complexes sur une architecture multi-cœurs 73

4.1 Spécification du problème . . . 74
4.2 Partitionnement de charges de travail à taux d'utilisation statique . . . 75
4.3 Partitionnement de charges de travail à taux d'utilisation dynamique . . . 75
4.3.1 Traduction du problème en système de contraintes linéaires . . . 76
4.3.2 Algorithme de recherche de solution optimale . . . 78
4.4 Ordonnancement des charges de travail . . . 82
4.4.1 Algorithmes généralement utilisés . . . 83
4.4.2 Ordonnancement avec *Earliest Deadline First* non-préemptif . . . 84
4.5 Résultats expérimentaux . . . 84
4.5.1 Méthodologie . . . 85
4.5.2 Evaluation de l'algorithme de *bin-packing* . . . 86
4.5.3 Recherche d'une solution valide . . . 87
4.5.4 Charges globales minimales atteintes . . . 88
4.5.5 Ordonnançabilité en fonction du taux d'utilisation unitaire . . . 93
Conclusion . . . 95

Conclusion 97

Contributions . . . 97
Perspectives . . . 98

Optimisation multi-critères de l'allocation 98
Prise en compte de l'hétérogénéité des cœurs 98
Analyse d'applications parallèles 99

A Programmes de test utilisés **101**

Table des figures **105**

Liste des tableaux **107**

Bibliographie **109**

Remerciements

Je voudrais tout d'abord remercier Bernard Goossens et Isabelle Puaut pour avoir accepté d'être les rapporteurs de mon travail. Leurs remarques constructives ont permis d'améliorer ce manuscrit.

Je remercie également Christian Percebois d'avoir accepté de participer au jury de ma soutenance de thèse.

Je remercie Christine Rochange et Pascal Sainrat pour m'avoir accueilli dans leur équipe et avoir guidé mon travail quatre ans durant, avec patience et expertise.

Je veux également remercier les autres membres de l'équipe TRACES, Hugues Cassé dont la maîtrise technique d'OTAWA n'a d'égal que sa disponibilité, Marianne de Michiel pour sa participation à mes travaux, ainsi qu'Armelle Bonnenfant avec qui j'ai eu l'occasion de collaborer dans mes activités d'enseignement. Je remercie Martine Labruyère pour le suivi administratif de ma thèse.

Je n'oublie pas mes collègues de bureau, Hajer et Haluk pour leur bonne humeur et leur gentillesse toujours égale. Ainsi que Cédric et Jonathan pour leur aide durant le début de ma thèse.

Je remercie sincèrement tous ceux, amis proches ou collègues doctorants qui ont été présents dans les instants de joie et dans les moments de découragement. Ceux qui ont su fermer les yeux sur mes indisponibilités et mes horaires de travail exotiques.

Enfin, je remercie du fond du cœur mes parents qui ont été mes plus grands supporters tout au long de ces années. Ce manuscrit clôt mon aventure étudiante, je le leur dédie.

Introduction

« Un système temps-réel est défini comme un système dont la conformité ne dépend pas seulement de l'exactitude des résultats, mais aussi du temps nécessaire à leur calcul. »

J. Stankovic (1988)

Les systèmes *embarqués* sont aujourd'hui présents partout dans notre environnement. De nombreuses tâches ayant trait à l'automatisation des machines, au confort, à la sécurité des êtres humains et des biens sont déléguées à ces systèmes autonomes de contrôle basés sur des capteurs et des calculateurs. La gestion du freinage dans les véhicules automobiles, le pilotage automatique des aéronefs ou la vidéo-surveillance des autoroutes sont autant d'applications possibles des systèmes embarqués dans notre vie quotidienne.

Ces systèmes se distinguent des autres équipements électroniques ou informatiques en ce que leur réactivité aux évènements extérieurs est primordiale. C'est pour cela que l'on parle de systèmes *temps-réel*, car la durée du traitement d'une information par le calculateur est une partie de la validité du résultat issu de ce traitement. Tout retard dans la lecture d'une donnée par un appareil de lecture vidéo peut entraîner une coupure de son ou d'image, tandis que tout retard dans la prise en compte d'un signal d'anomalie lors d'un lancement de fusée peut aboutir à sa destruction complète.

En conséquence, il est indispensable de connaître la durée d'exécution de l'*application* embarquée dans un calculateur temps-réel. Les applications sont généralement constituées d'une ou plusieurs tâches devant chacune respecter une échéance temporelle, c'est-à-dire une date avant laquelle finir l'exécution de leur traitement. La durée d'exécution d'une tâche n'est pas constante mais peut au contraire prendre un ensemble de valeurs différentes. C'est pourquoi valider le respect des contraintes de durée d'une application temps-réel nécessite d'effectuer une analyse de chacune des tâches afin d'estimer son *pire* temps d'exécution.

La majorité des calculateurs actuels sont basés sur des processeurs à l'architecture matérielle simple et à la rapidité limitée. L'utilisation de processeurs *multi-cœurs* est l'une des solutions envisagées afin d'augmenter le niveau de performances et l'étendue des fonctionnalités offertes par les systèmes embarqués de demain. Ces architectures où plusieurs *cœurs* de calcul sont intégrés sur la même puce réduisent effectivement les temps d'exécution des tâches. Cependant, de nombreuses ressources sont partagées entre les différents cœurs, ce qui occasionne des interférences entre les tâches et rend leur pire comportement temporel plus difficile à analyser.

Contributions

Ce document s'intéresse à l'analyse des délais d'accès pire-cas aux ressources partagées dans les plateformes multi-cœurs. Ce partage se fait généralement au moyen d'un *bus* d'interconnexion permettant aux cœurs de charger des informations depuis et vers des ressources communes de stockage, comme la mémoire centrale. Un *arbitre* contrôle et autorise les différentes demandes d'accès au bus. La *politique* de sélection des requêtes suivie par cet arbitre détermine les délais d'accès d'un cœur à une ressource partagée. Les politiques d'arbitrage *prévisibles* utilisées dans les plateformes multi-cœurs favorisent généralement la prévisibilité temporelle au détriment de l'adaptation aux besoins d'accès à la mémoire des tâches exécutées, ce qui nuit aux performances globales de la plateforme.

Nous proposons dans ce document un nouveau mécanisme d'arbitrage de bus *statique* établissant des niveaux de priorité différents pour chaque cœur, tout en garantissant une latence maximale pour chaque accès au bus afin de l'intégrer au calcul de temps d'exécution pire-cas. Notre méthode nécessite une analyse préalable des temps d'exécution des tâches. Elle permet de mesurer l'influence qu'ont les délais d'accès au bus sur les performances de ces tâches, et d'en déduire un niveau de priorité optimal pour chacune d'entre elles. Elle permet également de trouver une allocation optimale des tâches aux cœurs lorsqu'il y a plus de tâches à exécuter que de cœurs disponibles. Les résultats expérimentaux montrent une diminution significative des estimations de temps d'exécution pire-cas.

Organisation du document

Le chapitre 1 donne un aperçu général des approches utilisées pour l'estimation de temps d'exécution pire-cas sur une architecture mono-processeur, et plus particulièrement des techniques de l'analyse statique de tâches et de l'analyse de cache. Le chapitre 2 dresse un état de l'art des architectures multi-cœurs, du partage des ressources et de la prévisibilité temporelle des mécanismes arbitrant ce partage. Le chapitre 3 décrit deux nouvelles politiques d'arbitrage de bus permettant de garantir des niveaux de priorité différents pour chaque cœur et d'intégrer les latences d'accès obtenues dans le calcul de temps d'exécution pire-cas des tâches par analyse statique. Le chapitre 4 élargit le champ d'application de ces mécanismes à des plateformes exécutant une charge de travail complexe. Il présente une méthode d'allocation des tâches permettant de minimiser l'utilisation des cœurs tout en assurant l'ordonnancement de l'ensemble de tâches exécuté. Enfin, nous concluons ce document en discutant des problématiques soulevées par les résultats obtenus, et des perspectives de travaux futurs qui pourraient en découler.

Chapitre 1

Analyse de temps d'exécution pire-cas pour systèmes temps-réel strict

Les systèmes temps-réel sont des systèmes répondant aux évènements de l'environnement où ils sont déployés tout en respectant des contraintes temporelles précises. Le fonctionnement correct d'un système temps-réel dépend non seulement de la validité des résultats des calculs mais aussi de la garantie absolue que les temps d'exécution seront toujours respectés [91].

A titre d'exemple, le système de déclenchement des coussins de sécurité gonflables (ou *airbags*) d'un véhicule automobile est un système temps-réel qui protège les occupants en cas de collision. Le calculateur du système reçoit en permanence les mesures des capteurs accélérométriques placés à différents endroits du véhicule. Lorsqu'un évènement anormal est détecté, un algorithme intégré dans le calculateur calcule différents critères physiques, en fonction du type de collision (frontale, latérale, arrière), de la vitesse d'impact, ou encore de la raideur de l'obstacle. Ces critères déterminent l'utilité de déclencher ou non telle ou telle protection et à quel moment. Or pour être efficaces, les coussins gonflables doivent être déployés dans un délai de quelques dizaines de millisecondes après l'impact. La certification du programme intégré au calculateur doit donc garantir que son temps de décision est inférieur à ce délai, sous peine de ne pas protéger correctement les passagers du véhicule.

La défaillance d'un système temps-réel n'occasionne pas nécessairement de conséquences graves ou irrémédiables sur son environnement. On distingue les systèmes temps-réel *strict* (ou *dur*) des systèmes temps-réel *souple* par la criticité des applications exécutées : dans le premier cas aucun dépassement des contraintes temporelles n'est autorisé, alors que ceci est permis dans le second cas. En effet, les systèmes temps-réel souple sont généralement conçus pour tolérer une certaine proportion de retards dans l'exécution des tâches, tout en assurant une *qualité de service* (QoS) minimale. A l'opposé, le non-respect des échéances temporelles dans un système temps-réel strict entraîne son dysfonctionnement avec des conséquences potentiellement désastreuses (mise en danger de vies humaines, catastrophe écologique) comme nous l'avons vu dans l'exemple du système de déclenchement des coussins de sécurité. C'est pourquoi les dates de terminaison des tâches doivent être connues et *strictement* respectées.

Ce chapitre propose un aperçu des approches utilisées dans l'analyse temporelle de tâches exécutées sur une architecture de processeur mono-cœur classique.

1.1 Estimation du temps d'exécution pire-cas

L'exécution d'une tâche sur un système temps-réel peut prendre un certain nombre de valeurs de durée d'exécution différentes. Détecter et analyser le *pire* de ces scénarios en termes de temps d'exécution est le rôle de l'estimation de temps d'exécution pire-cas, en anglais *Worst Case Execution Time* (WCET). Déterminer cette valeur pour chacune des tâches exécutées permet de garantir que le système dans sa globalité respecte ses contraintes temporelles.

1.1.1 Définition de l'estimation de temps d'exécution pire-cas

Le temps d'exécution pire-cas d'une tâche est la *plus grande* durée d'exécution possible de cette tâche sur une même plateforme d'exécution cible.

Dans le cas où il est impossible de déterminer exactement cette *pire* durée d'exécution, on cherche à en trouver un majorant aussi peu élevé que possible. C'est l'objectif de l'estimation de temps d'exécution pire-cas. Le caractère critique des systèmes temps-réel implique que cette estimation doit impérativement être *sûre*, c'est-à-dire ne pas sous-estimer le pire temps d'exécution qui peut être observé. D'un autre côté, elle doit rester *précise* et ne pas surestimer cette durée de manière trop importante. Le dimensionnement de la plateforme d'un système temps-réel est lié aux durées maximales *estimées* des tâches qui sont exécutées dessus : toute surestimation trop importante de ces durées entraîne à son tour un sur-dimensionnement inutile du matériel.

1.1.2 Variabilité des temps d'exécution

Usuellement, la durée d'exécution d'une tâche peut prendre un ensemble de valeurs différentes. Ceci peut s'expliquer par la variabilité des données d'entrée de la tâche : lorsque ces valeurs sont fournies par des capteurs externes, toute modification de l'environnement du système embarqué peut occasionner des prises de décision ou des calculs supplémentaires.

L'architecture matérielle de la plateforme d'exécution du programme compte également parmi les facteurs qui expliquent la variabilité des temps d'exécution. Depuis l'apparition des premiers microprocesseurs dans les années 1970, leur performance n'a cessé de croître. Cette montée en puissance s'explique par deux facteurs.

Tout d'abord, l'amélioration des procédés de fabrication (et notamment de la finesse de gravure) a permis l'augmentation de la fréquence d'horloge des puces, c'est-à-dire la diminution de la durée d'un cycle d'horloge. Ceci est transparent pour l'estimation de temps d'exécution pire-cas lorsque l'architecture du processeur reste inchangée et que ce temps est exprimé en nombre de cycles processeur.

Ensuite, les architectes ont introduit dans les processeurs modernes un certain nombre de mécanismes micro-architecturaux tels que les caches, les pipelines ou la prédiction de branchement, visant à améliorer les performances et ainsi diminuer le temps d'exécution *moyen* des tâches. Ces mécanismes réduisent en effet le nombre de cycles nécessaires à l'exécution d'un programme, mais leur comportement est généralement dynamique, ce qui introduit une variabilité des scénarios d'exécution d'un même programme, et donc de

sa durée d'exécution. En conséquence, ils complexifient l'estimation précise d'un temps d'exécution pire-cas.

1.1.3 Approches et méthodes utilisées

Les approches dites *dynamiques* consistent à effectuer plusieurs exécutions successives de la tâche considérée et à observer leurs durées afin d'en déduire le temps d'exécution pire-cas de la tâche. La mesure de la *pire* durée d'exécution nécessite à la fois l'exécution de la tâche avec ses pires données d'entrées et avec l'état du matériel conduisant au pire comportement temporel du système. Ces critères étant difficiles voire impossibles à déterminer *a priori*, on utilise une quantité finie de jeux d'entrées et d'états auxquels la tâche peut effectivement être confrontée. Il n'est pas possible de garantir que ces scénarios couvrent effectivement le pire comportement temporel de la tâche. De plus, cette approche montre ses limites avec l'augmentation du nombre de variables d'entrée ou d'états possibles du système : elle provoque l'explosion du nombre de scénarios d'exécution à analyser.

Notre contribution fait partie des travaux basés sur l'analyse *statique* de programmes afin d'en estimer le temps d'exécution pire-cas. L'analyse statique désigne un ensemble de méthodes utilisées pour étudier le comportement d'un programme sur la seule base de son code, c'est-à-dire *sans avoir à l'exécuter*, ainsi que d'une description matérielle de la plateforme utilisée. C'est en cela que l'analyse statique se distingue des approches dynamiques, comme la mesure répétée de temps d'exécution. L'analyse statique d'un programme permet de prendre en compte l'influence qu'ont la structure du programme, les valeurs des données d'entrée ou les mécanismes micro-architecturaux du processeur sur les temps d'exécution.

Par la suite, nous dressons un récapitulatif rapide des différents outils et méthodes utilisés lors de l'estimation de temps d'exécution pire-cas, à la fois pour les approches dynamiques et statiques. Il est à noter que les valeurs obtenues par la mesure sont par définition toujours inférieures ou égales au temps d'exécution pire-cas *réel*, alors que l'analyse statique fournit des majorants plus ou moins proches de ce temps.

1.2 Approches dynamiques pour le calcul de temps d'exécution pire-cas

Les méthodes *dynamiques* mesurent le temps d'exécution pire-cas d'une tâche à partir d'un ensemble de jeux d'entrées. La mesure du temps d'exécution pire-cas suppose que l'on dispose au minimum du code exécutable (ou binaire) du programme, mais aussi souvent de son code source. Elle se fait directement sur la plateforme cible, ou à l'aide d'un simulateur utilisant un modèle de celle-ci.

1.2.1 Contexte

Il n'est pas toujours possible de réaliser des modèles matériels du système cible. Les spécifications complètes de l'architecture utilisée sont rarement disponibles, et son comportement temporel n'est généralement pas bien connu (bien qu'il existe maintenant certains projets de plateformes matérielles incluant dès leur conception des contraintes fortes

de prévisibilité temporelle, comme [32]). Dans ce cas, la mesure directe des temps d'exécution pire-cas sur le système cible permet de se passer d'une modélisation de la micro-architecture utilisée. Ceci permet aussi de garantir la pertinence des mesures réalisées, surtout si au lieu d'ajouter du code spécifique pour enregistrer les mesures l'on utilise un matériel dédié [90, 76].

Cependant, le matériel cible n'est pas nécessairement disponible, par exemple si le système est en cours de conception. Dans ce cas, la mesure des temps d'exécution nécessite un simulateur logiciel. Il doit modéliser précisément le matériel et garantir une exécution fidèle *au cycle près*. Les simulateurs permettent de mesurer plus facilement le temps des blocs de base de manière individuelle [11] car il n'est pas toujours possible de ne mesurer qu'une portion isolée du code sur le matériel cible. Ceci est utile pour les méthodes hybrides décrites dans la section suivante.

1.2.2 Stratégies d'obtention du pire scénario d'exécution

Déterminer le temps d'exécution pire-cas par la mesure nécessite de disposer du jeu de test qui conduit au temps d'exécution maximum. Différentes stratégies sont envisageables pour satisfaire cette exigence.

On peut d'abord utiliser tous les jeux d'entrée possibles. Pour un programme simple, l'utilisateur pourra peut-être fournir ces jeux de tests, en revanche pour un programme plus complexe il est nécessaire de les générer automatiquement. Williams *et al.* [106] proposent une méthode de génération de jeux de tests couvrant tous les chemins possibles d'un programme. Des techniques de génération automatique de jeux de données d'entrée par algorithme *génétique* [102, 10, 62, 67] sont également utilisées.

Les algorithmes *évolutionnistes* comme [104] s'appuient sur un ensemble de jeux de test de base, sur lequel est pratiquée une sélection des jeux satisfaisant au mieux les propriétés désirées (dans notre cas, les temps d'exécution les plus longs). Ceux-ci sont ensuite utilisés pour générer une nouvelle *population* par croisement et mutation. Ce procédé est ensuite appliqué itérativement.

Les algorithmes de *recuit simulé* comme [97] consistent à sélectionner les nouveaux scénarios candidats dans le *voisinage* d'un candidat courant. Les candidats répondant le mieux au problème sont toujours acceptés parmi les solutions. Quant aux autres, ils peuvent également être acceptés selon une heuristique prédéfinie, le but étant d'explorer l'espace des solutions de la manière la plus exhaustive possible. Il est possible d'accepter des candidats moins bons dans l'espoir qu'ils puissent conduire aux solutions optimales. Ceci est une autre manière de trouver des jeux de tests qui maximisent le temps d'exécution.

Enfin, certaines méthodes d'estimation de temps d'exécution pire-cas sont dites *hybrides*, car elles combinent à la fois les méthodes dynamiques avec des outils d'analyse statique de code [10, 30, 62, 105]. Dans ces approches, les mesures sur simulateur ou matériel cible peuvent être utilisées soit pour obtenir les temps d'exécution des blocs de base, soit pour déterminer les différents scénarios d'exécution de la tâche.

Les méthodes dynamiques et hybrides pour déterminer le temps d'exécution pire-cas d'un programme ont généralement un champ d'application limité. Hormis lorsque le scénario conduisant au temps d'exécution le plus élevé peut être déterminé aisément, c'est-à-dire pour des programmes simples, il est impossible de garantir totalement la perti-

nence des résultats obtenus. Les différents algorithmes et méthodes cités précédemment ne garantissent pas que les jeux de test générés mènent à l'obtention du temps d'exécution pire-cas. Ils peuvent seulement fournir une borne inférieure à la valeur réelle de ce temps comme [103], ce qui est inutile dans le cas de systèmes temps-réel strict. En revanche aucune des méthodes ne permet d'obtenir avec certitude un majorant du temps d'exécution pire-cas.

1.3 Analyse statique de temps d'exécution pire-cas

Afin de pouvoir déterminer un majorant du pire temps d'exécution d'une tâche sur un système temps-réel sans connaître *a priori* le jeu de test conduisant à ce résultat, des méthodes basées sur une analyse *statique* du code du programme ont été développées.

L'utilisation des méthodes de l'analyse statique implique que l'on ne peut pas prévoir *a priori* l'occurrence d'évènements extérieurs pouvant entraîner un délai supplémentaire dans l'exécution de la tâche, tels que la préemption d'une tâche par une autre, ou la survenue d'une interruption due à un autre composant du système. L'analyse statique d'une tâche ne reflète que son comportement temporel lorsqu'elle est exécutée en *isolation*, c'est-à-dire en l'absence de toute influence externe non prévisible. De plus, il est souvent difficile d'obtenir une prise en compte satisfaisante des effets temporels dûs à la complexité de la micro-architecture utilisée.

L'estimation de temps d'exécution pire-cas par analyse statique du code est un champ de recherche reconnu par la communauté temps-réel depuis une vingtaine d'années, et qui a bénéficié d'avancées importantes depuis les premières publications [82, 87].

L'analyse statique d'un programme se décompose généralement en trois phases. La première consiste en une analyse de flot ou analyse *de haut niveau* permettant de trouver les chemins d'exécution possibles dans le code du programme. L'analyse de *bas niveau* permet quant à elle d'intégrer les effets des mécanismes architecturaux du processeur dans les temps d'exécution des blocs de base. Enfin, le calcul final du temps d'exécution pire-cas combine les résultats obtenus dans les étapes précédentes. La décomposition de ces différentes étapes est détaillée dans la figure 1.1. Les nœuds ovales montrent les données intermédiaires produites par les différentes étapes d'analyse (représentées par des nœuds rectangulaires) qui mènent au résultat final.

1.3.1 Analyse de haut niveau

L'analyse de haut niveau permet de déterminer les différents chemins d'exécution possibles à l'intérieur d'un programme. Ce programme est pour cela décomposé en un ensemble de blocs de base. Un bloc de base est défini comme *une suite d'instructions séquentielles ne contenant qu'un seul point d'entrée et un seul point de sortie*.

Deux représentations sont généralement utilisées pour étudier le flot de contrôle d'un programme : l'arbre syntaxique ou le graphe de flot de contrôle (figure 1.3). Les exemples fournis reposent sur le code présenté dans la figure 1.2. La colonne de gauche correspond au code source de la fonction en langage *C* tandis que la colonne de droite contient un résultat possible de la compilation de ce code en langage *ARM*. Dans ce qui suit, nous

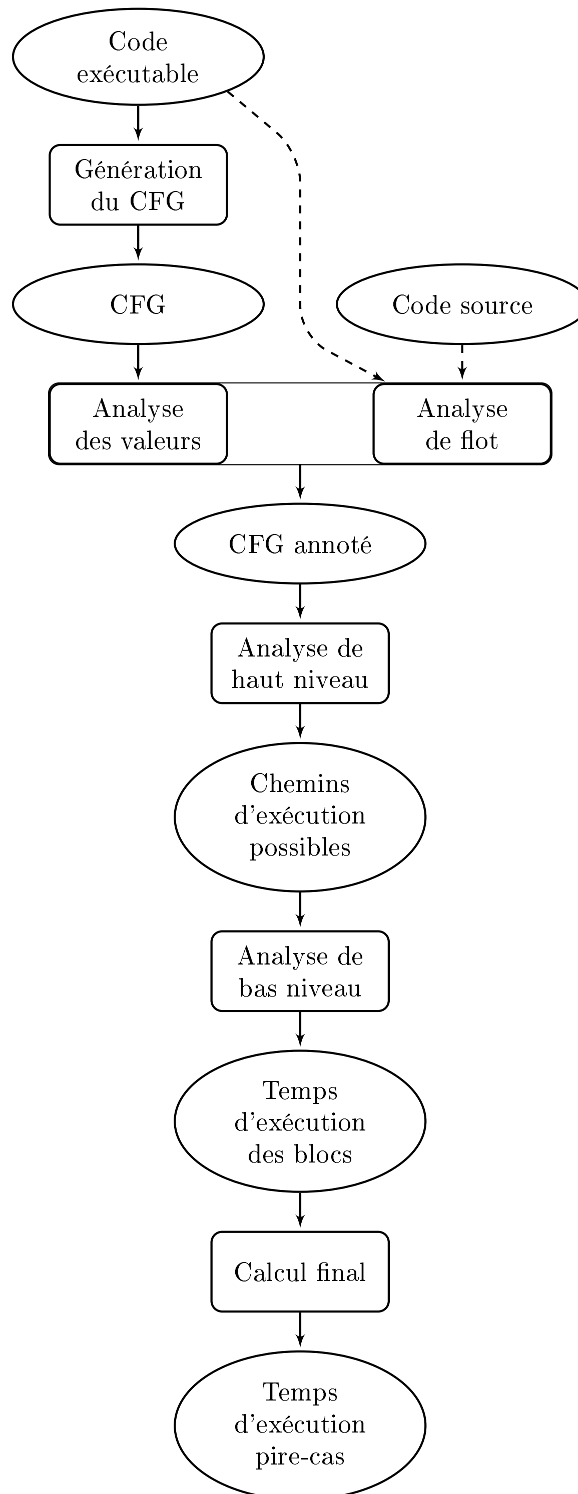

FIGURE 1.1 – Les différentes étapes du calcul de temps d'exécution pire-cas par analyse statique.

```
#define TAILLE 10

int somme(int *tab) {
  int i, sum;

  sum = 0;
  i = 0;
  while (i < TAILLE) {
    if (tab[i] < 0) {
      sum -= tab[i];
    }
    else {
      sum += tab[i];
    }
    i++;
  }
  return sum;
}
```

```
somme:                          Bloc0
 mov r0,#0        @sum
 adr r1,tab
 mov r2,#0        @i

boucle:                         Bloc1
 cmp r2,#10
 bge fin

 ldr r3,[r1,r2,lsl #2]          Bloc2
 cmp r3,#0
 bge else

 sub r0,r0,r3                   Bloc3
 b  suite

else:                           Bloc4
 add r0,r0,r3

suite:                          Bloc5
 add r2,r2,#1
 b  boucle

fin:                            Bloc6
```

FIGURE 1.2 – Code source et code assemblé de la fonction *somme*

détaillons les étapes du calcul de temps d'exécution pire-cas en utilisant les graphes de flot de contrôle.

Graphes de flot de contrôle

Le graphe de flot de contrôle est défini par un ensemble de blocs de base, et décrit les transitions possibles entre ceux-ci.

Les graphes de flot de contrôle sont construits à partir du code cible de la tâche à analyser. Le code est généralement en langage assembleur mais des travaux ont été menés sur le construction de graphes à partir du langage *Java bytecode* [9]. Certains outils de calcul de temps d'exécution pire-cas ou de manipulation de code bas niveau permettent de générer des graphes de flot, comme SALTO [85] ou OTAWA [16]. Certaines approches peuvent aussi utiliser un compilateur modifié à cet effet, comme dans [66]. Il est possible d'obtenir un graphe de flot de contrôle à partir du code source du programme, à condition que le compilateur ne fasse pas d'optimisation du code [80].

La figure 1.3 représente le graphe de flot de contrôle associé au code source de la fonction *somme* montré dans la figure 1.2. Deux types d'arcs orientés représentent les relations de précédence entre les blocs. Les branchements pris sont en pointillés, alors que l'exécution en séquence d'un bloc après un autre est représentée au moyen d'un arc plein. La fonction considérée se décompose en sept blocs de base, étiquetés $Bloc_0$ à $Bloc_6$. Le

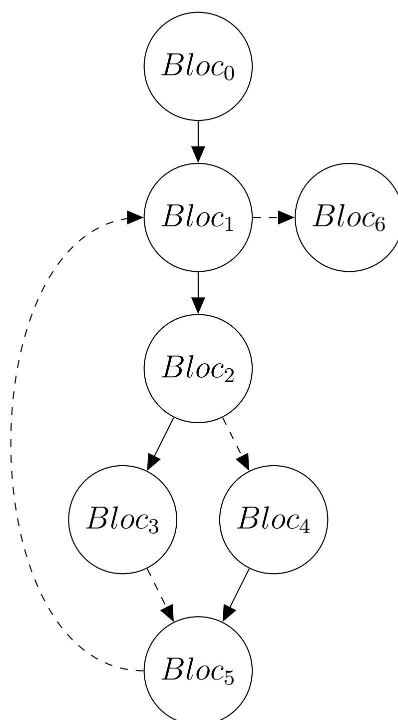

FIGURE 1.3 – Graphe de flot de contrôle de la fonction *somme*

point d'entrée de la fonction est le bloc $Bloc_0$. Le dernier bloc, $Bloc_6$ est un bloc vide qui constitue la sortie de la boucle.

Un graphe de flot de contrôle ne peut permettre à lui seul le calcul précis du temps d'exécution pire-cas de la tâche analysée, et doit être complété par des indications sur le flot du programme. En particulier, afin d'assurer que tous les chemins d'exécution soient de taille finie il est nécessaire de pouvoir définir des bornes supérieures sur le nombre d'itérations de chaque boucle. Ces informations peuvent être fournies à l'analyseur sous forme d'*annotations* [18, 75, 82], de constantes [41, 82] ou de manière interactive comme dans [51]. Il est également possible d'obtenir ces données automatiquement, comme dans [28]. L'analyse du flot de données peut être utilisée pour déterminer le nombre d'itérations des boucles et les chemins infaisables à l'intérieur de celles-ci [45].

De plus, certains chemins d'exécution apparaissent dans les représentations du programme mais ne peuvent être pris lors de son exécution réelle. Par exemple, deux branches du programme peuvent s'exclure mutuellement. Il faut donc représenter les chemins d'exécution infaisables afin de réduire le nombre de solutions à explorer pour la recherche du chemin d'exécution le plus long. L'identification de ces séquences de blocs de base peut se faire en modélisant le contenu des registres du processeur [35], ou en analysant l'évolution des valeurs des indices de boucle dans une représentation directe du code source [42].

L'outil *oRange* [27] permet d'extraire les bornes de boucle d'un programme en combinant l'analyse de flot et l'interprétation abstraite. Il prend en compte les appels non récursifs de fonctions, les conditions de boucles lorsqu'elles sont suffisamment simples, les *nids* de boucles (boucles imbriquées), et prend en compte la modification dynamique des valeurs des variables de boucle lorsqu'elles sont incrémentées par addition, soustraction ou multiplication. Pour chaque boucle rencontrée dans le code source, l'outil calcule le

$$
\begin{aligned}
x_0 &= 1 &&= x_{0,1} \\
x_1 &= x_{0,1} + x_{5,1} &&= x_{1,6} + x_{1,2} \\
x_2 &= x_{1,2} &&= x_{2,3} + x_{2,4} \\
x_3 &= x_{2,3} &&= x_{3,5} \\
x_4 &= x_{2,4} &&= x_{4,5} \\
x_5 &= x_{3,5} + x_{4,5} &&= x_{5,1} \\
x_6 &= x_{1,6}
\end{aligned}
$$

Table 1.1 – Système de contraintes généré par la méthode IPET.

nombre maximum d'itérations max ainsi que le nombre total d'exécutions du corps de la boucle $total$. La détermination de ces valeurs se fait selon les étapes suivantes. Premièrement, *oRange* identifie les boucles ainsi que leurs variables d'incrémentation et leurs expressions de sortie. Ensuite, il construit une représentation abstraite pour les variables max et $total$. Enfin, il en déduit une borne supérieure sûre. Les valeurs calculées par l'outil produisent une surestimation limitée par rapport aux valeurs réelles.

L'identification des chemins infaisables ou qui s'excluent mutuellement peut enfin se faire par *exécution symbolique* comme dans [4] ou [65]. Dans ces travaux, les tests de branchements permettent d'identifier les chemins qui ne sont jamais pris. Les techniques basées sur l'exécution symbolique peuvent être utilisées sur des langages spécifiques [64] ou des langages temps-réel [52]. Il est également possible de résoudre le problème d'une autre façon en dressant la liste des chemins faisables [3].

Calcul du plus long chemin avec la méthode IPET

L'obtention d'un graphe de flot de contrôle et des informations de flot qui lui sont liées permettent de déterminer le plus long chemin du programme associé en termes de durée d'exécution. Nous n'utiliserons que ce type de représentation de programme dans la suite de ce document.

La méthode de calcul du chemin menant au temps d'exécution pire-cas la plus utilisée dans les outils d'analyse statique est la méthode d'énumération des chemins implicites (*Implicit Path Enumeration Technique* ou IPET) [58]. Elle liste les différents chemins d'exécution à partir du graphe de contrôle de flot du programme, puis les formule en utilisant la *Programmation Linéaire en Nombre Entiers* (ILP : *Integer Linear Programming*). Un problème de PLNE consiste en une fonction à maximiser ou à minimiser sous des contraintes linéaires, ces contraintes étant exprimées à l'aide de variables entières. Le jeu de contraintes comporte deux parties : la première partie exprime la structure du graphe, la seconde partie modélise les informations de flot.

Une durée d'exécution (calculée par l'analyse de *bas niveau*) t_i et un nombre d'exécutions x_i sont associés à chaque bloc de base $Bloc_i$. La méthode IPET exprime chacune de ces données sous forme de contrainte linéaire dans le problème ILP. La table 1.1 liste les contraintes générées par la méthode d'énumération des chemins implicites pour le graphe de la figure 1.3. Le nombre d'exécutions du bloc i est égal au nombre d'exécutions des arêtes qui y mènent et au nombre d'exécutions des arêtes qui en sortent. Il est noté x_i

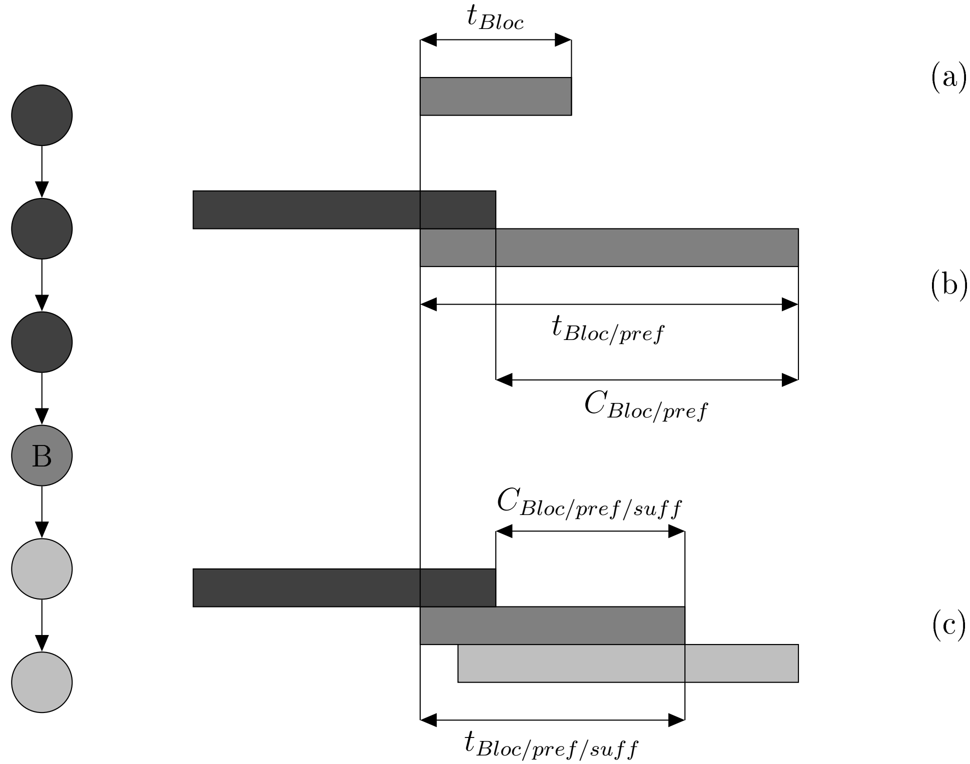

FIGURE 1.4 – Préfixe et suffixe d'un bloc de base.

dans le système de contraintes. Le nombre d'exécutions de l'arête effectuant la transition entre le bloc i et le bloc j est noté $x_{i,j}$. Dans la table 1.1, la contrainte $x_1 = x_{0,1} + x_{5,1}$ exprime le fait que le nombre d'exécutions du bloc 1 x_1 est égal à la somme des nombres d'exécutions de ses arcs entrants $(0, 1)$ et $(5, 1)$.

On rajoute ensuite les informations de flot extraites précédemment. Ici, le nombre d'itérations maximum de la boucle est modélisé par une nouvelle contrainte : $x_2 \leq x_{0,1} * 10$. Il est aussi possible d'inclure les paires de nœuds infaisables par des contraintes additionnelles [59]. Finalement, le temps d'exécution du programme est exprimé comme une somme de fréquences d'exécution des blocs de base assorties de leur temps d'exécution. Afin d'obtenir un majorant sûr du temps d'exécution pire-cas, il est nécessaire de maximiser cette expression au moyen d'un solveur ILP. Le temps de résolution est fortement corrélé à la complexité du système de contraintes [60].

1.3.2 Analyse de bas niveau

L'analyse de bas niveau permet de calculer les temps d'exécution de chaque bloc de base en utilisant le code compilé du programme. Elle est complémentaire de l'analyse de haut niveau (calcul du plus long chemin) et nécessite une modélisation précise de la microarchitecure utilisée. Les premiers travaux sur l'analyse statique de temps d'exécution pire-cas se sont d'abord focalisés sur des processeurs à la micro-architecture simple, où la durée d'exécution d'une instruction est constante. Le travail d'analyse de bas niveau se borne alors à utiliser le code compilé pour faire la somme des durées des instructions de

chaque bloc de base.

Cependant, les processeurs modernes intègrent de nouveaux mécanismes micro-architecturaux qui améliorent leurs performances au détriment de leur prévisibilité temporelle. Leur modélisation fine lors de l'étape d'analyse de bas niveau est primordiale afin d'améliorer la précision des estimations de temps d'exécution pire-cas. Nous détaillons ici la prise en compte de deux de ces mécanismes : les pipelines et les mémoires caches.

Exécution pipelinée

Plutôt que d'exécuter les instructions de manière purement indépendante, les processeurs utilisent un *pipeline* permettant le chevauchement (*overlap*) des exécutions de plusieurs instructions. Ceci permet d'atteindre théoriquement le débit d'une instruction exécutée par cycle processeur, voire plusieurs dans le cas des processeurs *superscalaires* mettant en œuvre le parallélisme d'instructions. L'exécution d'un bloc de base sur un processeur pipeliné doit tenir compte de diverses sources d'interférences entre instructions [89]. Une instruction peut être bloquée (*stall*) à un étage du pipeline en raison de l'indisponibilité de l'unité fonctionnelle constituant l'étage suivant : il s'agit d'un *aléa structurel*. Une instruction peut être bloquée car son exécution nécessite des données non encore mises à jour par une autre instruction : c'est un *aléa de données*. Enfin, une rupture de flot peut être induite par une instruction de branchement : il s'agit d'un *aléa de contrôle*. Toutes ces sources de délai supplémentaires doivent être modélisées.

Calculer les temps d'exécution de chaque bloc de base de manière individuelle ne signifie pas que ces blocs de code sont exécutés isolément les uns des autres. Au contraire, leur durée d'exécution est corrélée au contenu du pipeline avant l'exécution de la première instruction du bloc. Ce contenu dépend des instructions qui se sont exécutées avant elle, en d'autres termes, il dépend de l'historique des blocs de base exécutés précédemment, aussi appelé *préfixe*. Dans un processeur à ordonnancement dynamique des instructions (*out of order*), des dépendances de données et des conflits d'accès aux étages du pipeline peuvent aussi advenir avec le *suffixe*, c'est-à-dire les blocs de base suivant celui actuellement analysé.

La figure 1.4 illustre plusieurs scénarios d'exécution pour un même bloc de base. Dans le cas de figure (a), il est exécuté en isolation. Dans le cas (b), il a été précédé d'un préfixe, ce qui entraîne une modification de l'état du pipeline au début de l'exécution du bloc de base et une augmentation du temps d'exécution total du bloc $t_{Bloc/pref}$. Enfin, dans le cas de figure (c), ce même bloc est maintenant suivi d'un suffixe ce qui modifie encore son temps d'exécution. On observe que toute modification du préfixe ou du suffixe a un effet direct sur le temps d'exécution du bloc traité, noté $t_{Bloc/pref/suff}$. Pour être fiable, l'analyse temporelle de chaque bloc de base doit couvrir l'ensemble des couples préfixe/suffixe possibles. De plus, les effets temporels impactant le bloc courant peuvent être causés par des blocs exécutés bien avant lui. Ces phénomènes ont été identifiés sous le nom d'*effets longs* dans [33]. Il n'est pas possible de borner le nombre de blocs appartenant au préfixe et pouvant influer sur le bloc courant.

La figure 1.5 montre le chevauchement des exécutions des instructions dans un processeur pipeliné. Ceci induit que le temps d'exécution d'une suite $[Bloc_1 - ... - Bloc_n]$ de blocs est réduit par rapport à la somme des temps d'exécution des blocs de cette suite. Ceci se traduit par l'expression suivante :

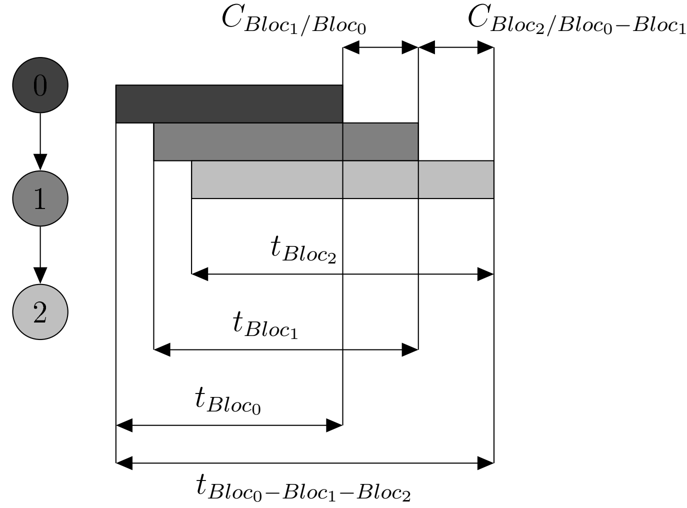

FIGURE 1.5 – Effets de recouvrement entre blocs de base.

$$t_{Bloc_0-...-Bloc_n} \leq \sum_{0 \leq i \leq n} t_{Bloc_i} \tag{1.1}$$

Par conséquent, il devient nécessaire de déterminer les *coûts* des blocs, c'est-à-dire la durée écoulée entre la fin de l'exécution de la dernière instruction du bloc précédent et la fin de l'exécution de la dernière instruction du bloc courant. Le coût C_{Bloc_i} du bloc $Bloc_i$ se calcule par rapport aux blocs le précédant ($Bloc_0 - ... - Bloc_{i-1}$). Le temps d'exécution de la séquence de blocs $Bloc_0 - ... - Bloc_n$ est la somme des coûts des blocs de la séquence :

$$t_{Bloc_0-...-Bloc_n} = \sum_{0 \leq i \leq n} C_{Bloc_i/Bloc_0..B_{i-1}} \tag{1.2}$$

Diverses techniques permettent de prendre en compte ces effets de recouvrement dans l'analyse de bas niveau. Il est notamment possible de représenter l'état du pipeline par des tables de réservation [50]. Cette méthode n'est efficace que dans le cas d'un pipeline scalaire et ne s'applique pas aux processeurs à ordonnancement dynamique des instructions. Il est aussi possible d'évaluer tous les états possibles du pipeline à l'aide des méthodes de l'interprétation abstraite. Cette technique est notamment utilisée par les outils aiT [1] et OTAWA grâce à la notion de graphes d'exécution [84].

Analyse du contenu des mémoires caches

Les processeurs embarquent généralement une quantité limitée de mémoire rapide appelée *mémoire cache* et assurant un rôle de stockage intermédiaire entre l'unité d'exécution et les données placées en mémoire principale. Le fonctionnement des mémoires caches est totalement transparent pour le programme exécuté. Elles peuvent être séparées en un *cache d'instructions*, contenant le code du programme à exécuter, et en un *cache de données*, ou bien regroupées en un seul *cache unifié*. L'intérêt d'utiliser des caches séparés

1. `http://www.absint.com/`

est de permettre un chargement simultané des instructions et des données. En outre, ceci facilite leur analyse temporelle.

Tout élément (instruction ou donnée) requis par le programme exécuté est d'abord recherché dans la mémoire cache. Si l'élément est trouvé, il s'agit d'un *succès* (*cache hit*) et il est envoyé au processeur, dans le cas contraire c'est un *défaut* de cache (*cache miss*) et l'élément est recherché dans le niveau hiérarchique suivant. Pour déterminer la latence de chaque chargement d'instruction ou de donnée du programme analysé, il est donc primordial de modéliser précisément l'état du cache, c'est-à-dire son contenu, afin de pouvoir prédire les éventuels défauts de cache. Une hypothèse conservatrice serait de considérer tous les accès mémoire comme des défauts de cache. Cela mènerait à une surestimation du temps d'exécution pire-cas. De plus, l'hypothèse selon laquelle un défaut de cache entraîne systématiquement un temps d'exécution plus élevé est mise à mal par l'existence d'anomalies temporelles [34].

Le contenu d'une mémoire cache à un instant donné dépend de ses caractéristiques techniques (nombre et taille des blocs), de l'historique des accès mémoire de la tâche exécutée mais aussi de la politique d'accès au cache utilisée. Dans un cache à accès direct, la mémoire est d'abord partitionnée en *plis* de la taille du cache, puis chaque pli est découpé en blocs de la taille d'une ligne de cache, de telle manière que le bloc n de chaque pli correspond à la ligne n du cache. Dans un cache associatif, chaque ligne peut accueillir n'importe quel bloc d'instructions ou de données lu dans la mémoire. Enfin, dans un cache associatif par ensembles, chaque ligne peut contenir des blocs issus de plis différents.

Les politiques d'accès associatives nécessitent de définir également une politique de remplacement des blocs dans le cache. Les politiques de remplacement les plus fréquemment utilisées dans les caches sont la politique FIFO (*First-In First-Out*), dans laquelle le bloc le plus *ancien* est remplacé en premier, ainsi que la politique LRU (*Least Recently Used*) qui évince le bloc de cache le moins *récemment utilisé* par le cœur. Ces politiques de remplacement nécessitent la présence de circuits logiques permettant de mémoriser l'*âge* des blocs présents dans le cache. Il existe des politiques de remplacement plus simples et moins coûteuses en termes d'implémentation, comme *Pseudo-LRU* [25].

L'analyse du contenu des caches d'instructions a fait l'objet d'un certain nombre de travaux, notamment [61, 36, 38, 69, 70, 8]. Ces méthodes supportent les caches directs et les caches associatifs qui implémentent une politique de remplacement de type LRU. Elles sont basées sur les techniques de l'interprétation abstraite [22] qui permettent de calculer des *états abstraits de cache* (*Abstract Cache States* ou ACS). Un ACS en un point de programme est une représentation de tous les contenus possibles de chaque ligne du cache et ceci pour tous les scénarios d'exécution possibles de la tâche. Établir l'ensemble des ACS permet de classer les accès aux blocs d'instructions en différentes catégories.

Un accès classé en tant que *Always Hit* signifie que c'est toujours un succès, *i.e.* l'instruction appartient à un bloc toujours présent dans le cache d'instructions. Au contraire, un accès classé en tant que *Always Miss* signifie qu'il y a systématiquement un défaut de cache et que la pénalité d'accès à l'élément suivant dans la hiérarchie mémoire s'applique. Des travaux ultérieurs ont introduit une troisième catégorie nommée *Persistent* ou *First Miss*. Elle s'applique généralement aux accès aux instructions qui appartiennent à un corps de boucle et restent présentes dans le cache entre les itérations successives de la boucle, mais peuvent entraîner un défaut de cache lors de la première itération. Plusieurs définitions et dénominations ont été proposées pour désigner les instructions appartenant

à cette catégorie [70, 96, 8]. Enfin, lorsque l'analyse n'est pas capable de déterminer un comportement sûr pour un accès donné, on l'étiquette comme étant *Not Classified*.

L'analyse du contenu des caches de données diffère quant à elle sensiblement de celle des caches d'instruction. L'unité d'exécution peut non seulement lire les blocs du cache, mais aussi remplacer leur contenu. De plus, les adresses des éléments à charger depuis la mémoire sont souvent calculés de manière dynamique par le programme. L'analyse doit aussi être en mesure de déterminer ces adresses. Ce calcul peut se faire par interprétation abstraite [37] ou en utilisant des techniques de simulation statique [50].

Conclusion

La connaissance des échéances temporelles des tâches exécutées est primordiale pour la sûreté des systèmes temps-réel strict. Les techniques de l'analyse statique décrites précédemment permettent de calculer le temps d'exécution pire-cas d'une tâche en isolation sur un processeur comportant un seul cœur d'exécution. Ces méthodes prennent en compte certains mécanismes micro-architecturaux des processeurs, comme les pipelines et les mémoires caches.

Cependant, l'utilisation récente de processeurs *multi-cœurs* dans les systèmes embarqués entraîne l'apparition de sources de délais supplémentaires, comme ceux occasionnés par les conflits d'accès aux ressources partagées entre les cœurs. Il est impératif de caractériser et de modéliser avec précision le comportement temporel de ces mécanismes externes au cœur d'exécution, afin d'intégrer leurs délais d'accès au temps d'exécution pire-cas de la tâche analysée.

Chapitre 2

Arbitrage, prévisibilité et calcul de temps d'exécution pire-cas pour architectures multi-cœurs

Encore plus que ceux utilisés dans des machines de bureau ou de calcul, les processeurs utilisés dans les systèmes embarqués ont des contraintes fortes en termes de coût, de consommation ou de dissipation thermique. De fait, les fréquences d'horloge utilisées sont généralement revues à la baisse, tandis que les jeux d'instructions et l'architecture sous-jacente sont simplifiés par rapport à ce qui est possible sur les processeurs classiques.

De nos jours, il existe une demande de plus en plus forte pour des systèmes embarqués disposant d'une puissance de calcul accrue, que ce soit pour des applications temps-réel *souple* (par exemple, acquisition à la volée de vidéos haute-définition sur un *smartphone*) ou des applications temps-réel *dur* (par exemple, système de détection de collision imminente dans un véhicule automobile).

Cependant, les architectures et techniques de gravure utilisées à l'heure actuelle par les principaux fondeurs limitent la montée en fréquence des puces à enveloppe thermique constante. Il est donc devenu primordial de trouver d'autres moyens d'améliorer les performances des processeurs embarqués. Les architectures parallèles, développées à l'origine pour répondre à des besoins accrus de performance dans les *clusters* de calcul et intégrées depuis sur une même puce physique, sont considérées depuis une décennie comme étant l'une des meilleures réponses à ce besoin. Ces architectures incluent deux types de processeurs distincts.

Les processeurs multi-flot simultané (*Simultaneous Multi-Threading* ou SMT) utilisent le parallélisme d'instruction afin d'exécuter conjointement plusieurs *threads* (processus légers) sur un même cœur *physique*. On parle alors de cœurs *logiques*.

Dans un processeur SMT, le but est d'optimiser le remplissage du flot d'instructions du processeur, ce qui permet d'améliorer sa vitesse de traitement. Ceci est généralement atteint en concevant des processeurs *superscalaires*, c'est-à-dire comportant plusieurs pipelines. L'efficacité du parallélisme de threads est d'autant plus assurée que ceux-ci sont conçus de manière à accepter un partage poussé des ressources. Le processeur doit également implémenter le partage des registres et des caches, ainsi qu'un système de suivi des instructions et données appartenant aux différentes tâches.

Les processeurs multi-cœurs (*Chip Multi-Processors* ou CMP) regroupent quant à eux

plusieurs cœurs *physiques*, le plus souvent identiques, sur un même composant. Ils sont reliés par un réseau d'interconnexion et partagent le plus souvent plusieurs étages de mémoire. [1] On ne doit pas les confondre avec les *systèmes sur puce* (*System on Chip*) qui sont des puces incluant tout un système embarqué (processeur, mémoires et contrôleurs d'entrée/sortie).

Dans ce second type d'architecture parallèle, le niveau de partage des ressources est moins important, puisque chaque cœur possède son propre pipeline, sa propre file de chargement d'instructions, ses propres caches de premier et aussi parfois de second niveau, et ainsi de suite. Au lieu de maximiser seulement le débit d'opérations calculées comme sur les processeurs SMT, ceci permet aussi d'améliorer les temps de réponse des différentes tâches, puisque leurs exécutions sont moins interdépendantes.

Cependant, les architectures CMP se heurtent à plusieurs facteurs limitant les performances et la prévisibilité de la plateforme qui n'existent pas sur les puces mono-cœur. Nous pouvons en lister deux principaux.

Premièrement, comme dans toute architecture parallèle, le niveau de performance atteint dépend de l'*homogénéité* des threads à exécuter. Dans un scénario idéal, toute tâche exécutée sur un processeur CMP devrait pouvoir être parallélisée en autant de processus légers qu'il y a de cœurs sur la machine, chacun de ces processus ayant un temps d'exécution identique et étant attribué à un cœur différent afin d'assurer une répartition optimale de la charge de calcul. Ceci est quasiment impossible à réaliser en pratique : la majorité des tâches comportent des portions de code non ou difficilement parallélisables. De plus, les synchronisations et communications entre processus entraînent des délais supplémentaires. Si toutes les tâches ne peuvent être parallélisées, l'ordonnanceur doit trouver une allocation des tâches aux cœurs qui puisse équilibrer idéalement la charge entre eux. Ceci est un problème d'optimisation que nous étudierons dans le chapitre 4.

Ensuite, les différents cœurs ne sont pas complètement indépendants les uns des autres. Afin de présenter des garanties sur l'intégration et de limiter le coût de développement et de fabrication des puces à plusieurs cœurs, un certain nombre de ressources sont *partagées* entre les cœurs d'exécution, comme la mémoire centrale, le bus permettant d'y accéder, ou encore les éventuels caches de second ou troisième niveau. Ce partage induit des conflits d'accès et des latences supplémentaires par rapport à une architecture mono-cœur. Il est nécessaire de déterminer une borne maximale précise de la valeur de ces latences, afin qu'elles soient incluses dans l'estimation de temps d'exécution pire-cas par analyse statique du programme. Ceci est possible à condition d'utiliser des mécanismes spécifiques de partitionnement et d'arbitrage des ressources partagées assurant le respect de contraintes temps-réel strict.

1. On peut noter que les technologies SMT et CMP sont implémentées et utilisées conjointement sur certaines microarchitectures, comme le *POWER7* d'IBM ou le *Nehalem* d'Intel.

2.1 Vue d'ensemble des différents types d'architectures multi-cœurs

2.1.1 Architectures multi-cœurs pour plateformes non mobiles

La conception des premières architectures multi-cœurs, basées en partie sur l'expérience acquise par les fondeurs sur les plateformes multi-processeurs, remonte aux années 90 et vise en premier lieu le marché de l'informatique *sédentaire*, c'est-à-dire les ordinateurs personnels, les serveurs, et les machines de calcul intensif qu'elles soient isolées ou en grappe (*cluster*).

On peut dissocier les architectures multi-cœurs en deux catégories. La première, majoritaire tant dans les machines *sédentaires* que dans les systèmes embarqués, rassemble plusieurs cœurs *homogènes*, c'est-à-dire identiques, et partageant certaines ressources sur une même puce. La seconde catégorie utilise des cœurs différenciés et spécialisés chacun dans des domaines différents : le traitement de l'image ou du son, le calcul pur ou l'ordonnancement global de la plateforme.

Parmi les processeurs multi-cœurs homogènes, le premier d'entre eux à avoir été commercialisé est le *POWER4* d'IBM en 2001. Il est constitué de deux cœurs basés sur une microarchitecture de type *PowerPC* et utilisant une architecture de jeu d'instructions (*Instruction Set Architecture*) 64 bits. La diffusion massive des processeurs multi-cœurs débute quelques années plus tard, en 2005, avec le lancement de deux modèles destinés au marché des ordinateurs de bureau et basés sur l'architecture *x86*. Le *Smithfield* d'Intel (plus connu sous l'appellation commerciale *Pentium D*) est conçu comme un assemblage de deux *Pentium 4 Prescott* gravés sur la même puce. Le *Manchester* d'AMD est quant à lui basé sur une nouvelle architecture pensée pour le multi-cœur : les cœurs partagent le même contrôleur mémoire et peuvent échanger des données entre eux sans avoir à utiliser le bus central. Cette caractéristique sera reprise plus tard par Intel avec la microarchitecture *Nehalem* [49].

Un bon exemple de puce pouvant être rangée dans la catégorie des processeurs multi-cœurs hétérogènes est le *Cell*, conçu dans le cadre d'un partenariat entre IBM, Sony et Toshiba. Son architecture est fortement optimisée pour le calcul parallèle : huit cœurs spécifiques, appelés *Synergystic Processing Elements* (SPE), sont destinés au traitement de toutes sortes de données numériques : image, vidéo, son ou encore résultats de simulation. Ils implémentent un jeu d'instruction *Single Instruction Multiple Data*, idéal pour des calculs lourds sur un grand nombre de données.

Ces huit cœurs sont contrôlés par un cœur principal appelé *PowerPC Processing Element* (PPE) qui est chargé d'allouer les tâches de calcul aux différents SPE et d'exécuter toutes les autres tâches. La communication entre PPE, SPE, mémoire centrale et périphériques d'entrée/sortie se fait par l'intermédiaire d'un bus d'interconnexion en anneau, appelé *Element Interconnect Bus* (cf. section 2.2.2). La différenciation des cœurs permet au *Cell* d'atteindre de meilleures performances qu'un processeur multi-cœur conventionnel. C'est ce qui rend l'utilisation de ce type d'architecture intéressante dans les *clusters* de calcul, les applications multimédia à fort besoin en puissance de calcul et bande-passante (haute-définition), ou encore l'imagerie médicale.

Les architectures multi-cœurs sont globalement une solution de rupture face à la course

à la fréquence d'horloge. L'organisation interne des cœurs est parfois modifiée en profondeur. Les pipelines longs comme celui de l'architecture *NetBurst* du *Pentium 4*, qui ne trouvent leur justification que dans la possibilité de monter en fréquence, voient leur nombre d'étages réduit.

2.1.2 Architectures multi-cœurs pour systèmes embarqués

Depuis plusieurs années, les fondeurs produisant des plateformes matérielles pour systèmes embarqués, à l'instar de Freescale, IBM ou Texas Instruments, ou encore les concepteurs tels que ARM s'intéressent au développement de nouvelles versions de leurs architectures implémentant plusieurs cœurs et destinées aux marchés de l'automobile, de l'aéronautique ou des machines industrielles.

Dans l'industrie aéronautique, une première étape préalable à l'utilisation de plateformes multi-cœurs est l'émergence depuis les années 1990 de l'*Avionique Modulaire Intégrée* (*Integrated Modular Avionics* ou IMA) [78], d'abord dans les appareils militaires puis dans un second temps sur les avions de ligne. Elle a pour but de grouper sur des calculateurs *modulaires* des tâches qui seraient autrement exécutées par des calculateurs séparés. Au lieu de développer des composants spécifiques pour chacun de ces calculateurs dédiés, l'IMA utilise des composants dits *pris sur étagère* (*Commercial Off-The-Shelf* ou COTS), c'est-à-dire fabriqués en grande série, de coût moindre et de puissance généralement supérieure. Ceci permet de diminuer drastiquement le poids et la quantité d'espace requise par l'avionique embarquée, ou d'utiliser des calculateurs identiques sur des modèles d'aéronefs différents.

L'exécution conjointe de plusieurs tâches temps-réel strict sur un calculateur modulaire suppose la division des ressources accessibles en sous-ensembles robustes. Ceci est assuré par l'utilisation de la norme *ARINC 653*. Elle sépare les tâches en différentes partitions qui ne doivent pas interférer entre elles. Les tâches de chaque partition partagent le même espace d'exécution et d'adressage mémoire. Ceci permet également de faire cohabiter des tâches de criticité différente sur un même calculateur. Cette technique, désormais fiabilisée est employée sur la plupart des nouveaux appareils mis sur le marché à partir de la fin des années 2000, comme l'*A380*, un avion gros-porteur du constructeur européen Airbus, ou le Boeing *787*.

La prochaine étape consiste en l'intégration de processeurs multi-cœurs dans les calculateurs des appareils de nouvelle génération, comme le projet *A30X*, un avion de ligne moyen-courrier qui sera vraisemblablement lancé dans une quinzaine d'années. L'utilisation d'architectures multi-cœurs dans les plateformes IMA a déjà fait l'objet de travaux, comme [39] ou [1]. La multiplicité des cœurs permet notamment d'isoler les exécutions des différentes partitions de tâches. En revanche l'accès aux ressources partagées entre ces partitions reste une source d'interférences et donc d'imprévisibilité temporelle.

2.2 Partage de ressources dans les architectures multi-cœurs

Le principe guidant généralement la conception et l'utilisation d'architectures parallèles est que certaines ressources doivent être communes aux différentes unités d'exécution.

Les processeurs multi-flot simultané entrelacent l'exécution de plusieurs tâches afin de maximiser l'occupation des unités fonctionnelles [99]. Les ressources partagées sur un processeur multi-flot simultané sont multiples : files d'instructions, unités fonctionnelles, caches et tables de prédiction de branchement.

Dans le cas d'une architecture multi-cœurs, les différentes unités d'exécution étant physiquement indépendantes, les ressources partagées sont celles qui sont externes aux cœurs mais communes à la plateforme, comme les caches, la(les) mémoire(s) principale(s), les bus ou les contrôleurs d'entrée/sortie.

On peut classer ces ressources en deux catégories [17] : celles qui gardent l'information pendant une durée plus ou moins importante, comme les mémoires, sont des ressources de *stockage*, alors que les ressources de *transit* changent d'état à chaque cycle d'horloge, à l'instar des mécanismes d'interconnexion.

2.2.1 Ressources de stockage

Dans un processeur multi-cœurs, chaque cœur dispose de son propre pipeline ou de ses propres unités arithmétiques et logiques. Le délai d'exécution des instructions faisant uniquement appel à ces ressources *privées* est donc strictement égal à celui observé sur un processeur mono-cœur. Mais comme nous l'avons vu précédemment, d'autres ressources sont *partagées* par l'ensemble de la plateforme, comme certains étages de la hiérarchie mémoire. Elles constituent une source majeure d'imprévisibilité temporelle pour les tâches qui y font appel, et sont par ailleurs un goulet d'étranglement pour les performances du processeur. Déterminer précisément le comportement temporel d'un accès mémoire est l'un des principaux défis lorsqu'on analyse une tâche exécutée sur une architecture multi-cœurs.

Principe d'une hiérarchie mémoire

Il est possible de tirer parti des phénomènes de localité *spatiale* ou *temporelle* afin d'accélérer les accès à la mémoire. L'utilisation d'une mémoire *hiérarchisée* permet de charger non pas seulement la donnée ou l'instruction requise par le processeur mais le bloc mémoire la contenant dans une mémoire intermédiaire plus rapide appelée *mémoire cache*. Ceci réduit les pénalités d'accès mémoire du processeur tout en conservant des possibilités de stockage importantes.

La plupart des architectures de processeur existantes, qu'elles soient utilisées dans un contexte mono-cœur ou multi-cœur, utilisent donc une hiérarchie mémoire structurée en plusieurs étages. Les composants de niveau $n+1$ sont généralement moins rapides mais offrent une plus grande capacité de stockage que ceux de niveau n. Dans la plupart des cas les différents niveaux de la hiérarchie mémoire sont inclusifs, c'est-à-dire que les données stockées au niveau n sont des copies d'un sous-ensemble des données stockées au niveau $n+1$.

Les premiers étages de la hiérarchie correspondent généralement aux mémoires caches de premier et second niveau, ultra-rapides et intégrées au processeur lui-même, viennent ensuite un éventuel cache de troisième niveau, une mémoire centrale à accès direct (*Direct Random Access Memory* ou DRAM) et enfin un périphérique de stockage permanent tel qu'une mémoire de masse magnétique (*disque dur*) ou utilisant des semi-conducteurs (mé-

moire *Flash*, *Solid-State Drive*). Ce principe de hiérarchisation des ressources de stockage se retrouve aussi bien sur les plateformes non mobiles que dans les systèmes embarqués.

Dans un processeur CMP, chaque cœur dispose de son (ses) propre(s) cache(s) de premier niveau. En revanche, les caches de second (et éventuellement troisième) niveau et la mémoire centrale sont généralement partagés entre les différents cœurs au moyen d'un mécanisme d'interconnexion. D'autres puces peuvent venir compléter l'espace d'adressage mémoire de l'unité d'exécution : on les appelle mémoires blocs-notes (*scratchpad*). Il s'agit de mémoires statiques (SRAM) ayant une latence d'accès faible et constante. Leur contenu est géré par voie logicielle et non par un mécanisme implanté matériellement sur le processeur. Leur taille ne permet généralement pas d'y stocker l'ensemble des instructions ou des données requises par le programme exécuté, c'est pourquoi des statégies optimales d'allocation des données dans la mémoire scratchpad sont recherchées [7].

Dans un processeur utilisant une hiérarchie mémoire, la lecture d'une instruction ou d'une donnée en mémoire suppose d'abord l'émission d'une requête de l'unité d'exécution du cœur vers son cache de premier niveau. Cette requête inclut l'adresse mémoire du mot demandé, ce qui permet de localiser la présence du bloc contenant le mot mémoire dans sa structure interne. En cas de succès du cache de premier niveau, l'accès mémoire ne dure que quelques cycles, alors qu'en cas de défaut, c'est la pénalité de lecture du bloc mémoire contenant l'élément requis dans la ressource suivante dans la hiérarchie mémoire (cache de niveau inférieur, mémoire principale) qui s'applique. Ceci étant, pour des raisons de performance, la recherche de l'élément se fait souvent en parallèle dans les différents niveaux de cache ou de mémoire.

La caractérisation du comportement temporel d'une hiérarchie mémoire est directement basée sur les résultats des méthodes d'analyse de cache de premier niveau vues en section 1.3.2. Certaines solutions reposant sur les techniques de l'interprétation abstraite permettent notamment de prendre en compte des mémoires caches à plusieurs niveaux [44]. Mueller a également formulé deux techniques afin de calculer les catégories des accès au cache de second niveau [68].

Analyse des conflits dans les caches partagés

Le partage de ressources de stockage entre cœurs soulève plusieurs problèmes. Premièrement, l'étendue de l'espace mémoire accessible et utilisable par un cœur dépend de la quantité de blocs déjà occupés par des instructions ou des données liées à un autre cœur. Cependant lorsqu'on estime le temps d'exécution pire-cas d'une tâche en isolation, il est *impératif* de connaître la fraction *utilisable* de la capacité des éléments de stockage à l'avance. Si aucun mécanisme de restriction d'accès aux mémoires n'est mis en place, la ou les tâches exécutées par un cœur peuvent accéder à l'espace mémoire d'un autre cœur et en corrompre le contenu. Ceci met également à mal l'analyse statique du programme, puisque les techniques d'interprétation abstraite ne savent pas prendre en compte les effets *destructifs* causés à la ressource de stockage par des accès extérieurs.

Plusieurs travaux récents se proposent d'inclure ce phénomène de corruption des données dans l'analyse du contenu des mémoires caches. L'analyse des caches est initialement réalisée pour chaque tâche en isolation à l'aide des méthodes de l'interprétation abstraite. Par la suite, on réalise une seconde passe pour chaque tâche, en supposant que tout bloc de cache également utilisé par une autre tâche que la tâche analysée peut voir son contenu

corrompu par la tâche tierce. Les accès à ce bloc sont donc classés comme étant *Always Miss* pour toutes les tâches [108]. Dans le cas d'un cache associatif par ensembles, les éventuels conflits peuvent mener à une modification de l'âge des blocs [57, 43].

Le pessimisme de cette approche peut engendrer une grande surestimation du temps d'exécution pire-cas au fur et à mesure que le nombre de tâches concurrentes devient élevé et que ces tâches ont une occupation importante du cache de second niveau. Il est possible de minimiser cet effet en analysant l'ordonnancement de l'ensemble de tâches, afin que les tâches qui accèdent aux mêmes blocs de cache mais qui ne sont pas exécutées simultanément ne soient pas considérées comme conflictuelles [57]. Une autre solution, proposée dans [43], est de forcer l'évitement du cache de second niveau pour les blocs qui ne sont chargés qu'une fois et qui ne bénéficient donc pas du stockage en cache. Ceci réduit le nombre de conflits possibles.

Afin d'éviter tout recouvrement d'un bloc de cache ou de mémoire partagée par une autre tâche, il est également possible de *partitionner* physiquement ou logiquement les espaces d'adressage de chaque tâche, ou de *verrouiller* à l'avance le contenu des mémoires. La méthode du partitionnement est utilisée dans [77] pour simplifier l'analyse de cache sur des processeurs mono-cœur exécutant plusieurs tâches. Dans le même type de configuration, [79] propose plutôt de verrouiller le contenu des lignes de cache. Lorsque ce contenu peut être modifié à la volée par un algorithme prenant en compte le comportement des tâches exécutées, le verrouillage est *dynamique*, dans le cas contraire il est *statique*.

L'utilité de ces différentes méthodes dans un processeur multi-cœurs a été récemment étudiée dans [93], qui mesure les effets du partitionnement des caches, combiné à leur verrouillage statique ou dynamique. Il est soit possible de réserver un espace en mémoire pour chaque tâche, soit le partitionnement peut se faire en fonction des cœurs, c'est-à-dire que n'importe quelle tâche peut utiliser la totalité de la partition allouée au cœur sur lequel elle s'exécute. Les résultats montrent que la seconde possibilité, couplée à un verrouillage dynamique du cache, est celle qui améliore le plus les estimations de temps d'exécution pire-cas. Il existe en outre plusieurs stratégies d'allocation des partitions de mémoire cache aux cœurs. Il semble plus efficace d'allouer une ou plusieurs unités physiques (*bancs*) de cache à chaque cœur, plutôt qu'une ou plusieurs voies dans un cache associatif par ensembles [74].

2.2.2 Mécanismes d'interconnexion

Dans une plateforme multi-cœurs, les mécanismes d'interconnexion revêtent une grande importance. Il est en effet nécessaire d'éviter tout goulet d'étranglement lors des échanges entre les différentes unités d'exécution et les ressources qui leur sont communes. Tout sous-dimensionnement entraîne l'explosion des latences d'accès aux ressources partagées, ce qui peut empêcher une tâche de respecter ses échéances temporelles et mettre à mal l'ordonnançabilité de l'ensemble de tâches exécuté.

La topologie de l'interconnexion utilisée a également des répercussions sur la consommation électrique du système hôte, sur la surface de la puce et donc sur son coût de production. A titre d'exemple, le mécanisme d'interconnexion nécessaire à un processeur octo-cœurs peut consommer autant d'énergie qu'un cœur, occuper autant de transistors que trois cœurs, et ajouter une pénalité équivalente à la moitié de celle d'un accès au cache de niveau 2 [54].

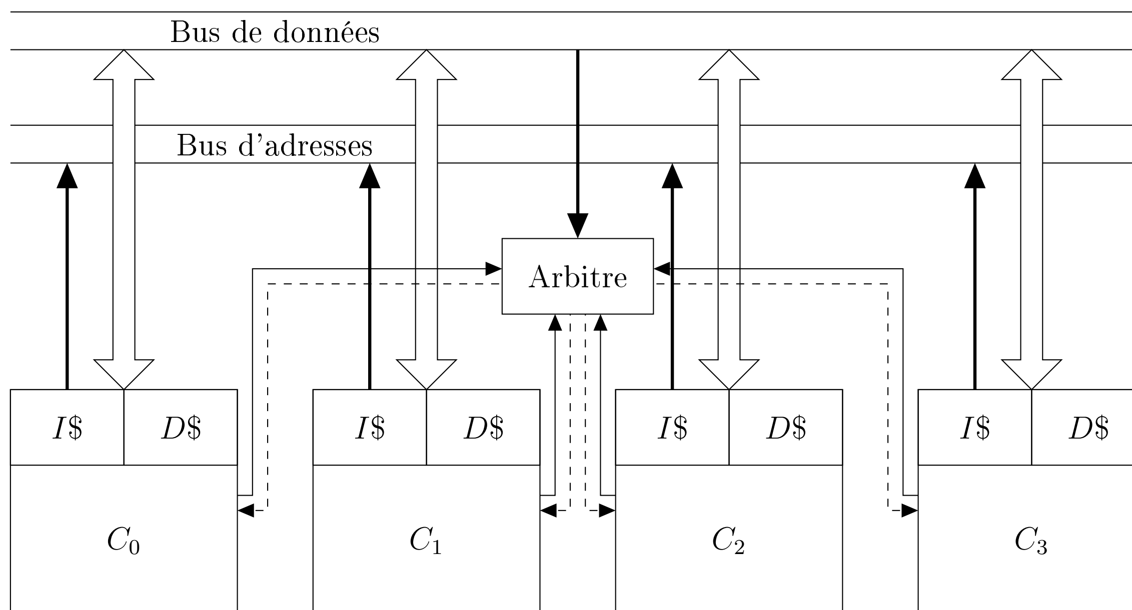

FIGURE 2.1 – Représentation schématique d'une plateforme multi-cœurs interconnectée par un bus partagé.

Un élément d'interconnexion bien conçu et adapté limite le besoin d'une mémoire cache de niveau 3 [49]. Les mémoires caches étant de grandes consommatrices d'espace sur les puces multi-cœurs, ceci permet d'utiliser les transistors libérés afin d'implanter des unités d'exécution supplémentaires. De plus, enlever un niveau de cache permet de réduire le pessimisme de l'analyse statique et améliore la précision des estimations de temps d'exécution pire-cas.

Dans ce qui suit, nous décrivons trois topologies d'interconnexion largement utilisées dans les architectures multi-cœurs actuelles : le *bus*, le *crossbar* et l'*anneau*.

Bus partagé

Sur un processeur multi-cœurs, le *bus* est la topologie d'interconnexion la plus répandue entre les différents éléments de la plateforme : cœurs, caches, mémoire partagée, interfaces d'entrée/sortie. L'utilisation des bus afin de faire communiquer des unités d'exécution entre elles n'est pas nouvelle car ils sont déjà largement répandus dans les architectures multi-processeurs [107]. L'intégration poussée et le partage accru des ressources dans les multi-cœurs induit des volumes de communications encore plus importants entre les différents éléments, ce qui nécessite l'utilisation de lignes de bus plus longues et pipelinées afin d'améliorer le débit des données transmises. Or, la requête émise par un élément doit traverser l'intégralité du bus pour être finalement visible par tous les autres éléments. La propagation de la requête pourra donc prendre plusieurs cycles.

Par la suite, nous admettons qu'il est possible de borner à une valeur fixe et constante le temps de propagation d'une requête sur le bus, ainsi que la pénalité d'un accès en mémoire quel que soit l'emplacement de la donnée et l'élément qui y accède. La figure 2.1 décrit la structure et le fonctionnement d'une plateforme multi-cœurs à 4 unités d'exécution et utilisant un bus partagé. Chaque cœur est doté d'une unité d'exécution, d'un

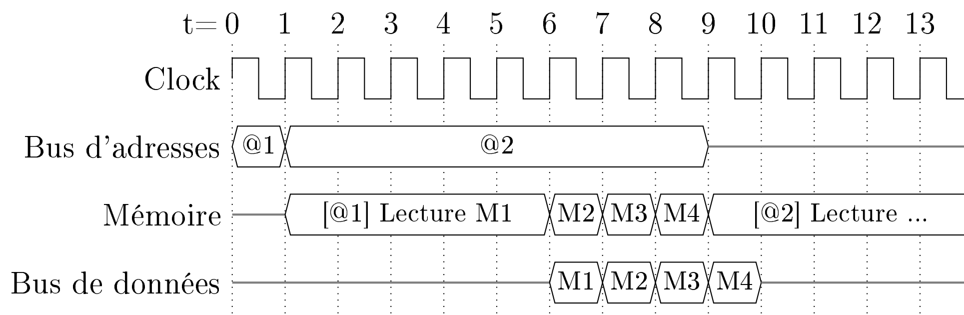

FIGURE 2.2 – Diagramme temporel d'une opération de lecture en mémoire.

cache d'instructions et d'un cache de données séparés. Idéalement, il dispose aussi d'un contrôleur de bus utilisant le mode DMA (*Direct Memory Access*) et permettant ainsi de décharger l'unité d'exécution de la gestion des transferts sur le bus. Les autres ressources sont externes et partagées avec les autres cœurs. Les requêtes entrantes sont stockées dans une file (non représentée ici) puis servies par un arbitre de bus.

Lorsqu'une donnée ou une instruction est requise par une tâche, le contrôleur de bus du cœur sur lequel s'exécute cette tâche envoie une *requête* d'accès à l'arbitre. Ce signal *request* est représenté sur la figure par une flèche noire pleine et fine. L'arbitre traite les différentes requêtes suivant une politique d'arbitrage qui lui est propre. La sélection de la requête à satisfaire en premier peut se faire suivant sa date d'arrivée, la priorité statique du cœur qui l'a émise, ou encore suivant le principe du *tourniquet*, c'est-à-dire que les cœurs sont servis à tour de rôle. Un cœur ne peut accéder au bus que lorsque l'arbitre lui en a *accordé* l'accès. Ce signal *grant* est représenté sur la figure par une flèche en pointillés. Le cœur envoie une commande de lecture ou d'écriture sur le bus de commande (non représenté ici). Il présente l'adresse en mémoire du bloc à lire ou à écrire sur le bus d'adresses. Ceci est représenté sur la figure par une flèche noire pleine et épaisse. S'il s'agit d'une écriture, il transmet également les données à envoyer sur le bus de données, en un ou plusieurs cycles. Les transferts depuis et vers le bus de données sont représentés par des flèches bidirectionnelles blanches et larges. Ce bus est lui-même également bidirectionnel, grâce à l'agrégation de deux bus unidirectionnels [54].

La figure 2.2 présente le séquencement d'une opération de lecture en mémoire d'un bloc de 4 mots à travers un bus. Les différentes lignes de bus et la mémoire sont synchronisés par une même horloge. Le bus de données a la largeur d'un mot mémoire. L'envoi d'une commande de lecture et la propagation de l'adresse sur le bus d'adresse débute au cycle $t = 0$. Au cycle suivant, la mémoire reçoit la commande et l'adresse du ou des mots à lire. La lecture d'une donnée en mémoire est une opération complexe qui nécessite la sélection de la ligne puis de la colonne à charger [12]. Ces opérations durent chacune plusieurs cycles d'horloge, dans le cas présent la lecture du premier mot prend cinq cycles. Il est ensuite transféré sur le bus de données au cycle suivant. Les trois mots suivants sont déjà pré-chargés dans le cache interne de la mémoire (*row buffer*), leur lecture ne prend donc qu'un seul cycle. Dans le cas d'une mémoire centrale partitionnée, plusieurs cœurs peuvent accéder simultanément à des bancs de mémoire différents. En revanche, le transfert des blocs sur le bus de données reste séquentiel.

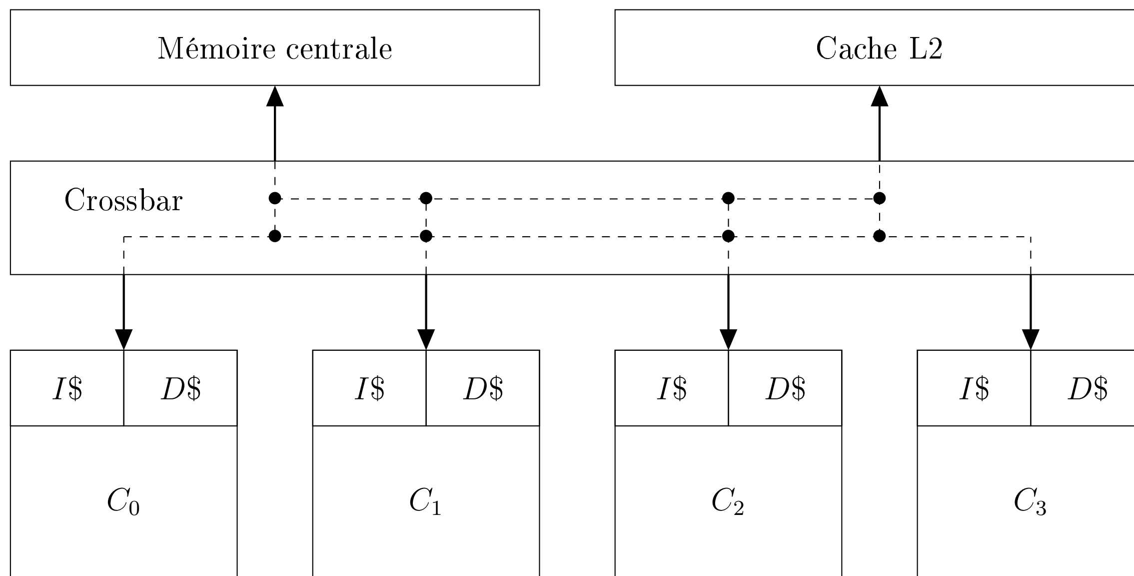

FIGURE 2.3 – Représentation schématique d'une plateforme multi-cœurs interconnectée par un réseau multi-étages.

L'augmentation du nombre d'éléments à interconnecter est une problématique majeure dans la conception des plateformes multi-cœurs embarquées basées sur un bus conventionnel, comme AMBA [5] ou CoreConnect [21]. Elles reposent sur des modèles transactionnels où les éléments de calcul commandent eux-mêmes les opérations de lecture et d'écriture, lesquelles sont ensuite divisées en plusieurs transactions qui doivent être arbitrées séparément. En l'absence d'une politique de partage du bus adaptée, les délais peuvent exploser en cas de forte charge (cf. section 2.3).

Crossbar et interconnexions point-à-point

Les mécanismes d'interconnexion à barres croisées, plus communément appelés *crossbars* permettent de faire disparaitre le phénomène de goulet d'étranglement qui peut être observé lorsqu'un bus partagé est saturé de requêtes. Ils sont basés sur un principe de partitionnement partiel ou total des communications. Dans un crossbar complètement connecté, chaque nouvel élément doit être doté d'autant de bus de communication qu'il y a d'éléments distants avec lesquels il doit échanger des données. En fonction du dimensionnement des points d'accès aux ressources de stockage partagées, il est même possible que plusieurs requêtes soient servies au même cycle d'horloge sans interférer entre elles ni avec des requêtes ayant des émetteurs ou des destinataires différents. Pour ces raisons un crossbar est bien plus performant qu'un bus classique. En revanche son implémentation nécessite une multiplication des lignes logiques utilisées. Ceci est supportable dans une architecture multi-cœurs simple. En revanche, dans une architecture massivement parallèle comprenant de nombreux éléments à faire communiquer, le coût de l'implémentation ainsi que l'espace occupé sur la puce peut se révéler prohibitif.

Il est possible de contourner la complexité de mise au point du crossbar en utilisant à la place une implémentation non complète telle que le réseau multi-étages. La figure 2.3

illustre un exemple de plateforme multi-cœurs architecturée autour d'un réseau multi-étages. Les lignes pleines représentent la liaison entre chaque élément et l'interconnexion. A l'intérieur de l'interconnexion, les éléments sont reliés entre eux par un maillage de lignes directes ou bien indirectes, c'est-à-dire passant par un ou plusieurs *commutateurs* (représentés ici par des points). Pour n éléments à relier, le crossbar complètement connecté possède n^2 commutateurs tandis que le réseau multi-étages n'en a que $n.log(n)$ [24] : dans l'exemple que nous présentons, il est ainsi possible de faire communiquer les 6 éléments avec seulement 4 commutateurs. Ces mécanismes d'interconnexion point-à-point ont fait leur apparition sur les processeurs généralistes (*QuickPath Interconnect* chez Intel [49], *HyperTransport* chez AMD [20]). Chaque élément accédant au réseau forme un nœud et est connecté à un ou deux autres nœuds de l'interconnexion qui relayent les paquets émis. Cependant, les éléments ne peuvent pas communiquer directement entre eux : leurs paquets sont d'abord routés par un module hôte (*Host Bridge*). Ceci permet à la fois de déterminer le chemin le plus court pour chaque requête et d'éviter plus facilement les conflits d'accès. A l'instar des réseaux traditionnels, ce type d'architecture est évolutif et peut s'adapter à des plateformes intégrées (*System on Chip*) ayant un grand nombre d'éléments à interconnecter. Cependant, l'utilisation d'un réseau point-à-point réduit la bande passante garantie entre les éléments à cause des interférences entre requêtes. Il semble préférable d'utiliser un autre type d'interconnexion permettant de libérer des transistors et d'implémenter de nouveaux cœurs d'exécution [54].

Anneau

Les topologies de bus en anneau (*ring bus*) combinent plusieurs avantages : elles sont plus compactes, plus simples à développer et à implémenter qu'un crossbar, tout en restant plus performantes qu'un bus classique. La figure 2.4 illustre le fonctionnement d'un anneau. Chaque élément à relier au réseau est muni d'un nœud (*stop*) sur l'anneau. A l'instar d'un réseau multi-étages, les paquets passent donc par plusieurs nœuds avant d'être délivrés à leur destinataire. A la manière d'un périphérique routier ces interconnexions sont le plus souvent constituées de deux anneaux, l'un fonctionnant dans le sens horaire et l'autre dans le sens anti-horaire, ce qui permet aux paquets d'emprunter le chemin le plus court. Dans cette topologie bi-directionnelle, un paquet de données n'a jamais besoin de parcourir plus de la moitié de l'anneau pour atteindre l'élément de destination. Ceci permet à plusieurs transactions d'avoir lieu simultanément, à condition qu'elles n'utilisent pas les mêmes portions des lignes logiques [13].

Le bus en anneau est utilisé dans l'architecture *Sandy Bridge* d'Intel [88]. Il s'agit d'une évolution du *Nehalem* auquel il a notamment été ajouté des éléments supplémentaires à interconnecter : une unité de traitement graphique, une unité de compression/décompression vidéo matérielle et des cœurs additionnels. Ici, l'utilisation d'un crossbar se heurte à des contraintes de coût et d'occupation physique de la puce : chaque connexion entre un nouvel élément de calcul et le cache de troisième niveau nécessite un millier de lignes logiques supplémentaires. Afin de résoudre ce problème, le crossbar utilisé dans l'architecture précédente a été remplacé par une topologie basée sur quatre boucles indépendantes. Le premier anneau sert à transférer les données (*data ring*), le second à propager les signaux de requête (*request ring*), le troisième sert à renvoyer les acquittements (*acknowledge ring*), enfin un dernier anneau sert à gérer la cohérence des caches (*snoop ring*). Chaque

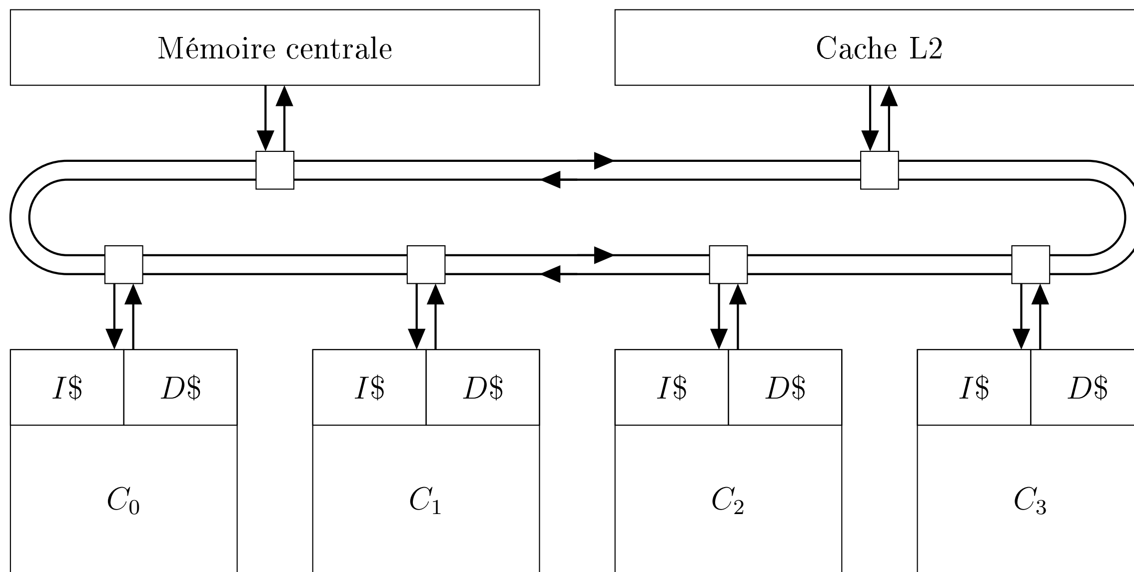

FIGURE 2.4 – Représentation schématique d'une plateforme multi-cœurs interconnectée par un bus en anneau bi-directionnel.

interconnexion (*stop*) entre un élément et un anneau permet de faire transiter 256 bits de données par cycle d'horloge.

Dans le processeur *Cell*, les communications entre le cœur principal (PPE), les unités de calcul (SPE), la mémoire principale et les périphériques d'entrée/sortie se font également au moyen d'un bus en anneau appelé *Element Interconnect Bus* ou bus EIB. Il est composé de quatre boucles d'une largeur de 128 bits. Au contraire d'un crossbar, il n'est pas coûteux ni compliqué d'ajouter un nouvel élément à un bus en anneau. C'est pourquoi il semble être une topologie adaptée à un système massivement parallèle où un nombre élevé d'éléments partagent des ressources communes.

2.3 Prévisibilité du comportement pire-cas des mécanismes d'interconnexion

Nous venons de voir que les communications entre éléments dans un processeur multi-cœurs peuvent se faire par l'intermédiaire de différentes topologies de bus : les pénalités subies par un cœur lors de l'accès à un élément partagé dépendent directement du type d'interconnexion utilisé. Lors de l'analyse statique il est difficile de prévoir les délais dus à l'interconnexion, qui peuvent parfois dépendre du comportement des autres cœurs (préemption de la ressource) ou même reposer sur des mécanismes d'acheminement des requêtes non modélisables.

A titre d'exemple, les bus en anneau ne sont pas à notre connaissance utilisés dans des plateformes ayant des contraintes temporelles strictes. Sur un réseau multi-étages ou point-à-point, le chemin emprunté par un paquet de données peut varier en fonction de la politique de routage utilisée, ou en fonction de facteurs externes à la requête (par

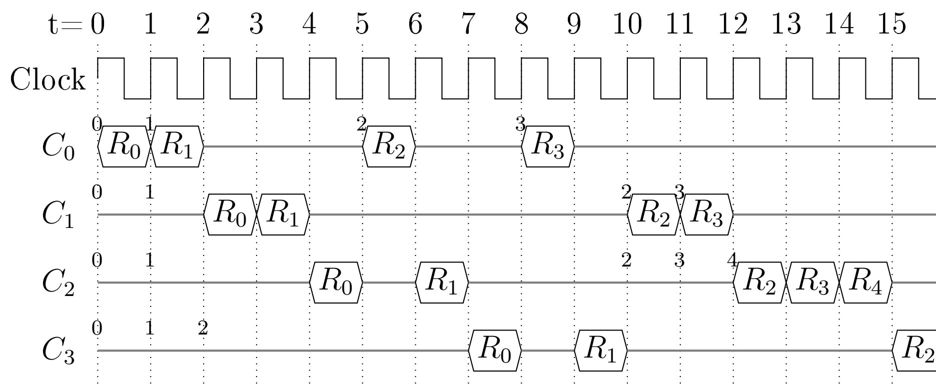

FIGURE 2.5 – Exemple d'arbitrage de bus avec un ordonnanceur à priorités fixes.

exemple, l'occupation d'un commutateur par une autre communication). De plus, la mise à l'échelle d'un réseau point-à-point induit que les paquets doivent traverser un nombre de plus en plus important de nœuds, ce qui nécessite des algorithmes de routage plus complexes. Ces algorithmes étant d'abord conçus pour améliorer le débit moyen du réseau et non sa prévisibilité, la détermination des latences pire-cas des requêtes est rendue plus difficile [53].

Dans la suite de cette section, nous ne traitons que des bus classiques, qui sont des mécanismes d'interconnexion beaucoup plus largement répandus dans les plateformes multi-cœurs dédiées à l'embarqué, comme le Freescale MPC8641D [2]. Les politiques de sélection des requêtes suivies par les arbitres de bus peuvent ici être classées en deux catégories principales. La première comprend des politiques cherchant à maximiser la bande passante globale du bus, c'est-à-dire à assurer de bonnes performances *moyennes*. Au contraire, les politiques appartenant à la seconde catégorie visent à prévoir une performance *minimale* pour chaque accès, ce qui permet de calculer une borne maximale sur le délai notée par la suite *UBD* (*Upper Bound Delay*). Cette valeur doit être intégrée à l'analyse statique des tâches exécutées sur la plateforme afin d'assurer la robustesse des estimations de temps d'exécution pire-cas et le respect des dates d'échéance.

2.3.1 Protocoles et mécanismes d'arbitrage de bus non prévisibles

Arbitrage à priorités fixes

De nombreux systèmes informatiques n'ayant pas de contraintes temps-réel et utilisant des ressources partagées fonctionnent avec un simple ordonnanceur à priorités. Un niveau de priorité est préalablement attribué à chaque élément accédant au bus, et l'arbitre se base sur ce paramètre pour gérer l'ordonnancement des requêtes. La figure 2.5 présente l'exemple d'une séquence de requêtes sur un bus partagé par quatre cœurs et ordonnancé par un arbitrage à priorités. Afin de simplifier le chronogramme, on considère que chaque accès au bus dure un cycle. Le niveau de priorité de chaque cœur est inversement proportionnel à son numéro, *i.e.* $Prio(C_0) > Prio(C_1) > Prio(C_2) > Prio(C_3)$.

2. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8641D`

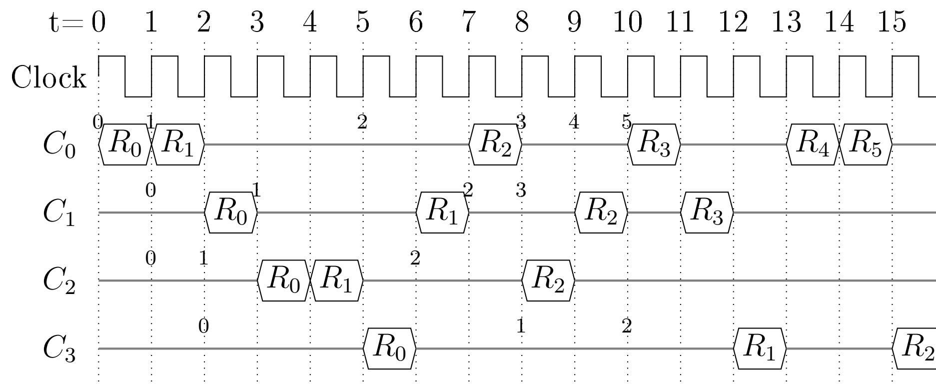

FIGURE 2.6 – Exemple d'arbitrage de bus avec la politique *First-Come First-Served.*

Pour chaque cœur, l'indice $i \in [0, n[$ est positionné sur la date d'arrivée de la i-ème requête émise par le cœur et le bloc R_i sur le cycle où cette requête est effectivement transmise sur le bus. Si comme au cycle $t = 0$ plusieurs demandes d'accès au bus sont en concurrence ou en attente, c'est celle issue du cœur ayant la priorité la plus haute (ici, C_0) qui est d'abord satisfaite. Ce type d'arbitrage est facile à implémenter et est donc largement utilisé. Cependant, il est impossible pour un élément émettant une requête de prévoir quel délai il devra attendre avant que celle-ci soit servie. Une requête peut être retardée indéfiniment par des éléments de priorité supérieure. Seules les latences de l'élément ayant la plus haute priorité sont prévisibles.

Ceci n'est pas suffisant si le système comprend plusieurs tâches critiques concurrentes s'exécutant sur des cœurs différents. En conséquence, un ordonnanceur à priorités ne peut être utilisé de manière sûre sur une plateforme multi-cœurs que lorsqu'un seul de ces cœurs exécute une tâche ayant des contraintes temps-réel strictes. La latence pire-cas d'accès au bus est alors nulle pour ce cœur ($UBD_0 = 0$, ou 1 s'il n'y a pas de préemption de bus). A l'inverse, elle ne peut être déterminée pour les autres cœurs [56]. De plus, ce schéma d'arbitrage simple ne prend pas en compte le comportement des tâches exécutées par les différents éléments, ce qui peut nuire à la performance globale de la plateforme.

Arbitrages à priorités dynamiques

Afin de pallier à cet inconvénient, il est courant d'utiliser des politiques où les niveaux de priorité sont attribués en fonction des dates d'arrivée des requêtes, comme le protocole *First-Come First-Served* (FCFS). Les requêtes sont stockées dans une file, ce qui permet de les traiter en fonction de leur date d'arrivée. Plus une requête est ancienne, plus tôt elle sera satisfaite. Si plusieurs requêtes arrivent au même moment, la sélection de celle qui sera placée en première position dans la file peut être aléatoire ou bien suivre la politique du tourniquet. Ce schéma de fonctionnement est illustré par la figure 2.6.

On observe que même si le cœur C_0 émet plus de requêtes que les autres cœurs dans le même intervalle de temps, tous bénéficient d'un traitement équitable qui consiste à traiter les requêtes les plus anciennes en priorité. Un cœur qui émet fréquemment des requêtes a plus souvent accès au bus que les autres cœurs, mais sans bloquer les requêtes des autres éléments, ce qui permet d'optimiser l'occupation du bus partagé et d'améliorer les

performances de la plateforme [83]. En revanche, au cycle $t = 8$ l'arbitre reçoit plusieurs requêtes simultanées, mais le cœur C_3 qui a envoyé moins de requêtes que les autres n'est pas favorisé pour autant. Le protocole FCFS n'inclut pas de mécanisme de *mémorisation* qui permettrait d'établir une priorisation des requêtes arrivant en même temps suivant l'historique du cœur les ayant émises. Le délai de satisfaction de n'importe quelle requête dépend uniquement du nombre de requêtes arrivées précédemment et non encore servies. Prévoir ce délai nécessiterait une analyse conjointe de toutes les tâches exécutées par la plateforme, afin de déterminer les dates de toutes les requêtes.

Certaines variantes, comme *First-Ready First-Come First-Served* (FR-FCFS) [100, 46] tentent d'améliorer FCFS en prenant aussi en compte la localité spatiale des accès. Les requêtes à la ligne de mémoire active sont traitées en priorité, ce qui évite au contrôleur mémoire de charger une nouvelle ligne à chaque fois, et réduit donc les latences d'accès mémoire pour ces requêtes. Le corollaire de ce mécanisme est que lorsqu'une tâche génère un flux d'accès consécutifs à une même ligne de mémoire, les requêtes des autres tâches peuvent être mises en attente pour des durées importantes [71]. De même que dans FCFS, les requêtes les plus anciennes sont toujours prioritaires, ce qui désavantage les tâches qui émettent moins de requêtes que les autres.

Afin de corriger les défauts de la politique FR-FCFS, Nesbit *et al.* [72] proposent un ordonnanceur *équitable* basé sur des concepts d'arbitrage utilisés dans les applications réseau afin de garantir un niveau élevé de qualité de service (QoS). L'espace mémoire de chaque tâche est représenté sous la forme d'un espace d'adressage *virtuel* auxquels les délais d'accès ont une valeur constante. La requête à satisfaire par l'ordonnanceur est sélectionnée suivant ce paramètre : il faut qu'elle ait lieu avant sa date d'échéance *virtuelle*. Ceci permet d'offrir la QoS demandée. La bande passante non consommée est distribuée en priorité aux tâches ayant le moins accédé au bus. L'approche présentée permet d'améliorer la bande passante du bus et l'équité entre tâches.

Mutlu *et al.* [71] ont proposé un protocole configurable destiné à améliorer l'équité de traitement des méthodes basées sur les dates d'arrivée des requêtes, sur une plateforme où les différents cœurs sont également en compétition pour l'accès à la mémoire centrale. Pour chaque tâche exécutée par chaque cœur, l'ordonnanceur met en permanence à jour deux compteurs : T_{shared} exprime le total du nombre de cycles passés par la tâche à attendre le bus (*stall time*), tandis que T_{alone} représente la valeur estimée de ce temps d'attente si la tâche s'exécutait en isolation, comme sur une architecture mono-cœur. Le rapport entre ces deux valeurs représente le *ralentissement dû à la mémoire*. L'ordonnanceur implémenté sélectionne en priorité les requêtes provenant de tâches ayant un facteur de ralentissement élevé. Ceci permet d'équilibrer les temps d'attente entre les différentes tâches, et donc d'améliorer les performances du système.

L'ensemble des mécanismes d'arbitrage que nous venons de voir ont pour but de maximiser l'utilisation du bus. Les plus complexes d'entre eux permettent une gestion dynamique des priorités en essayant d'instaurer un arbitrage équitable entre tous les cœurs. Mais les latences de bus d'une tâche critique évoluent toujours dynamiquement en fonction du comportement des autres tâches exécutées par le système. Il n'est pas possible de déterminer une borne fixe des latences vues par un cœur par la seule analyse de la tâche qu'il exécute. En outre, les mécanismes présentés ne sont souvent pas déterministes, ce qui empêche leur modélisation abstraite.

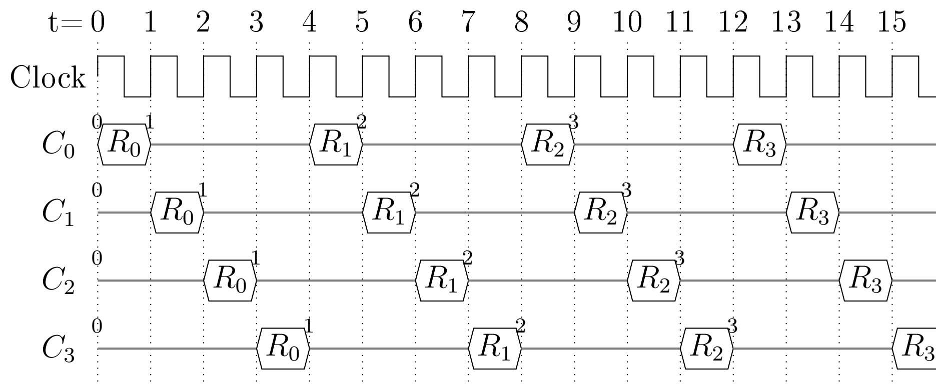

FIGURE 2.7 – Exemple d'arbitrage de bus avec la politique du tourniquet.

2.3.2 Mécanismes d'arbitrage de bus compatibles avec l'estimation de temps d'exécution pire-cas

Arbitrage des accès par la politique du tourniquet

Certains protocoles d'ordonnancement traitent les requêtes de chaque cœur de manière à ce que l'on puisse borner les délais dûs aux conflits avec les requêtes des autres cœurs. La plus simple de ces politiques d'arbitrage est celle du *tourniquet* (*Round Robin*). Dans la politique du tourniquet, chaque élément interconnecté peut accéder au bus à tour de rôle et pendant le même quantum de temps. En conséquence, la latence maximum d'une requête émise par un cœur est fonction du nombre de cœurs qui partagent le bus, et de la durée d'un accès au bus. Ce délai est le même pour chaque cœur, est prévisible et ne dépend pas de la nature des tâches exécutées sur les autres cœurs [74]. La figure 2.7 donne un exemple de fonctionnement *pire-cas* de la politique du tourniquet. Les quatre cœurs émettent quatre requêtes successives chacun. Ces requêtes sont servies à tour de rôle par l'arbitre de bus. Le délai (*distance*) maximum entre deux requêtes sur le bus issues du même cœur est exprimé par la formule suivante :

$$UBD = (N - 1) \times L_{bus} \tag{2.1}$$

Dans l'équation (2.1), N représente le nombre de cœurs exécutant des tâches temps-réel strict, et L_{bus} la latence d'accès au bus. Dans l'exemple présenté dans la figure 2.7, la valeur de ce délai est de 3 cycles.

Avec la politique du tourniquet, même si une tâche effectue plus de requêtes sur le bus, par exemple parce que son code ne tient pas en cache, il n'est pas possible de donner une priorité d'accès au bus plus élevée au cœur exécutant cette tâche. Tous les accès au bus subissent la même latence, ce qui peut éventuellement dégrader les performances de la plateforme si celle-ci exécute un ensemble de tâches *hétérogène*, c'est-à-dire n'ayant pas les mêmes besoins en bande passante.

Ordonnancement statique des accès

Les politiques d'ordonnancement *statiques*, de type *Time Division Multiple Access* (TDMA) semblent être de plus en plus utilisées dans les systèmes embarqués. De la même manière que dans *Round-Robin*, tous les cœurs sont servis à tour de rôle, mais pas nécessairement dans l'ordre. L'occupation du bus par un élément est représenté par un créneau (*slot*) ayant généralement la longueur d'une requête. Tout ordonnancement du bus est composé d'un certain nombre de slots qui sont alloués à l'un ou à l'autre des cœurs. Il est possible pour un cœur d'utiliser plusieurs slots successifs. L'ordre d'allocation est déterminé hors-ligne, comme dans [6] et [101]. Lors de l'exécution, l'arbitre répète l'ordre d'allocation périodiquement en attribuant aux différents cœurs les slots d'occupation du bus qui leur reviennent.

Rosén *et al.* [86] proposent une solution d'arbitrage prévisible basée sur TDMA. Ils définissent un graphe de tâche pour chaque tâche exécutée sur la plateforme, où les périodes de calcul sont clairement séparées des périodes de communication entre tâches. Dans chaque graphe, les nœuds représentent les séquences de calcul pur à l'intérieur des tâches, tandis que les arêtes représentent les dépendances de données entre ces tâches (communications explicites). Les dates d'échéance de chaque tâche, et si possible de chaque séquence de calcul sont ajoutées au graphe sous forme d'annotations. L'échange de données entre des tâches allouées à des cœurs d'exécution différents se fait par l'intermédiaire d'une mémoire partagée accessible via un bus. Dans ce cas, les requêtes d'accès au bus sont ordonnancées en fonction des slots alloués à chaque cœur. La séquence d'allocation des slots est écrite dans une table d'ordonnancement stockée elle-même dans une mémoire directement connectée à l'arbitre de bus.

Cependant, lors de l'analyse statique il est impossible de savoir si les requêtes sont synchronisées aux slots qui leur correspondent [92]. La solution proposée par [86] afin de déterminer si un accès au bus se produira durant le slot qui lui est réservé est de calculer la date de début de chaque nœud dans le graphe de flot de contrôle. Ceci se fait lors de l'analyse de bas niveau des différentes tâches. Dans le cas d'une tâche ayant plusieurs chemins d'exécution, ce qui est presque toujours le cas, il est nécessaire de dérouler tous ces chemins dans le graphe de flot de contrôle, ce qui n'est pas compatible avec les méthodes d'estimation de temps d'exécution pire-cas par analyse statique.

En conséquence, le calcul robuste de la valeur maximale de la latence d'accès au bus (UBD) exige d'additionner la longueur de tous les slots alloués aux autres cœurs. Si les slots ont la longueur d'un accès au bus, ceci revient à utiliser la politique du tourniquet ; dans le cas contraire, les latences d'accès au bus sont dégradées, ce qui peut mener à des résultats exagérément pessimistes. C'est pourquoi TDMA n'est pas une solution satisfaisante dans le contexte de l'analyse statique de temps d'exécution pire-cas [109]. Cette méthode n'a d'intérêt que dans le cas de tâches simples, dont le code peut n'être composé que d'un seul chemin d'exécution, par exemple à l'aide de la méthode décrite dans [81].

Staschulat *et al.* [92] cherchent à dissocier les requêtes mémoires issues de tâches temps-réel strict de celles qui ne le sont pas. Dans le premier cas, le cœur ne peut souffrir une latence élevée, et celle-ci doit pouvoir être bornée. C'est notamment le cas lorsqu'on cherche à résoudre un défaut de cache. En revanche, ce type de requête ne demande pas une bande passante élevée, étant donné qu'elle ne porte que sur une quantité restreinte de données (un bloc de cache), et n'advient que périodiquement. Dans le second cas, on

n'a pas à respecter des échéances temporelles strictes, et on cherche plutôt à maximiser la performance moyenne des accès. Un cœur peut cependant exiger une bande passante élevée à un instant donné (*burst*). Wandeler *et al.* [101] présentent une méthode permettant de déterminer la taille minimum des slots TDMA alloués à des tâches respectant des échéances temporelles strictes.

Les ordonnanceurs à *budget* [92, 2] permettent également de satisfaire des besoins en bande passante différents. Cependant, au lieu d'établir un ordonnancement de bus statique comme dans TDMA, ils allouent simplement une quantité donnée de bande passante à chaque cœur. En fonction de son niveau de priorité, et tant qu'il n'a pas consommé sa quantité allouée (budget) de slots de bus, un cœur peut voir sa requête satisfaite immédiatement.

La manière dont ces mécanismes pourraient être modélisés afin d'être pris en compte dans le calcul de temps d'exécution pire-cas n'est pas claire. Généralement, prédire les latences implique ici de connaître la durée séparant deux requêtes au bus, en effectuant une analyse préalable de la tâche. Comme nous l'avons vu précédemment, ceci est réalisable pour certaines applications, par exemple lorsqu'on étudie les accès aux données pour une application de flux de données, mais se révèle beaucoup plus complexe dans le cadre d'accès dynamiques et irréguliers à la mémoire, comme ceux requis pour remplir le cache d'instruction lors de défauts de cache. En effet, les analyses basées sur des représentations abstraites du contenu du cache ne peuvent généralement pas déterminer le comportement de tous les accès au cache d'instructions [8].

Conclusion

Les différentes unités d'exécution d'une architecture multi-cœurs partagent un certain nombre de ressources de stockage, ainsi qu'un mécanisme d'interconnexion permettant d'accéder à ces ressources. Toutes les politiques d'arbitrage utilisées sur ces mécanismes d'interconnexion ne peuvent pas être prises en compte dans l'estimation de temps d'exécution pire-cas par analyse statique. Certaines ont été conçues pour maximiser l'utilisation du bus afin d'améliorer les performances du système. Cependant elles ne sont pas adaptées aux systèmes temps-réel critiques, car elles ne permettent pas de borner les latences d'accès au bus. D'autres sont prévisibles, mais leur fonctionnement nécessite de modéliser l'entrelacement des requêtes des différents cœurs, ce qui implique également d'effectuer une analyse croisée de l'ensemble des tâches exécutées par la plateforme. Afin de pouvoir établir une synchronisation des différentes requêtes, ces tâches devraient suivre un chemin d'exécution défini à l'avance où les temps d'exécution des blocs de base sont constants, ce qui est très improbable en réalité.

Enfin, les politiques statiques et indépendantes du comportement des tâches sont à la fois prévisibles et analysables mais dégradent les performances des tâches exécutées en imposant une latence d'accès au bus fixe et uniforme pour tous les cœurs. Or les ensembles de tâches exécutés par un système embarqué n'ont pas nécessairement les mêmes besoins en bande passante : les tâches dont le code tient en cache ou qui bénéficient d'une forte localité spatiale effectuent moins de requêtes aux ressources communes de stockage que les autres. Considérer une équité absolue de traitement entre les différents cœurs amène à une surestimation importante des délais d'accès au bus et à un sur-dimensionnement

du mécanisme d'interconnexion nécessaire. Notre but est de proposer un protocole adapté à des ensembles de tâches *hétérogènes*, qui puisse fournir un niveau de bande passante différencié entre les cœurs tout en respectant des contraintes de criticité maximales. Ce mécanisme doit être déterministe et permettre le calcul des délais maximum d'accès au bus, afin de pouvoir être modélisé et intégré au calcul de temps d'exécution pire-cas par analyse statique.

Chapitre 3

Arbitrage de bus multi-niveaux pour des charges de travail hétérogènes

Comme nous l'avons vu précédemment, les mécanismes d'arbitrage complètement indépendants du comportement des tâches, comme la politique du tourniquet, ne permettent pas d'établir une hiérarchisation entre les cœurs exécutant des tâches effectuant beaucoup d'accès aux ressources de stockage partagées, et les autres. Cette équité de traitement peut empêcher certaines de tâches de respecter leurs dates d'échéance, alors que si elles étaient favorisées par le mécanisme d'arbitrage, l'ordonnançabilité du système s'en trouverait grandement améliorée. Dans le cas d'applications *parallèles* séparées en plusieurs tâches s'exécutant simultanément, il peut également être intéressant de diminuer les latences subies par les tâches critiques, *i.e.* celles qui provoquent le plus d'attente aux points de synchronisation. Les politiques d'ordonnancement statique des accès au bus comme *Time Division Multiple Access* ne sont quant à elles pas adaptées à l'analyse statique de temps d'exécution pire-cas. Pourtant, ce sont les seules permettant de prendre en compte la spécificité des ensembles de tâches *hétérogènes*, ayant des besoins en bande passante différents.

Ce chapitre présente un nouveau mécanisme d'arbitrage permettant d'offrir différents niveaux de priorités aux différentes tâches exécutées par une plateforme multi-cœurs. Il est basé sur un partitionnement du bus système indépendant du comportement des tâches exécutées. L'allocation d'une tâche à l'un ou l'autre des niveaux de priorités doit être réalisée avec le plus grand soin. L'objectif recherché est d'améliorer les performances globales de la plateforme cible, tout en maintenant les latences d'accès au mécanisme d'interconnexion prévisibles.

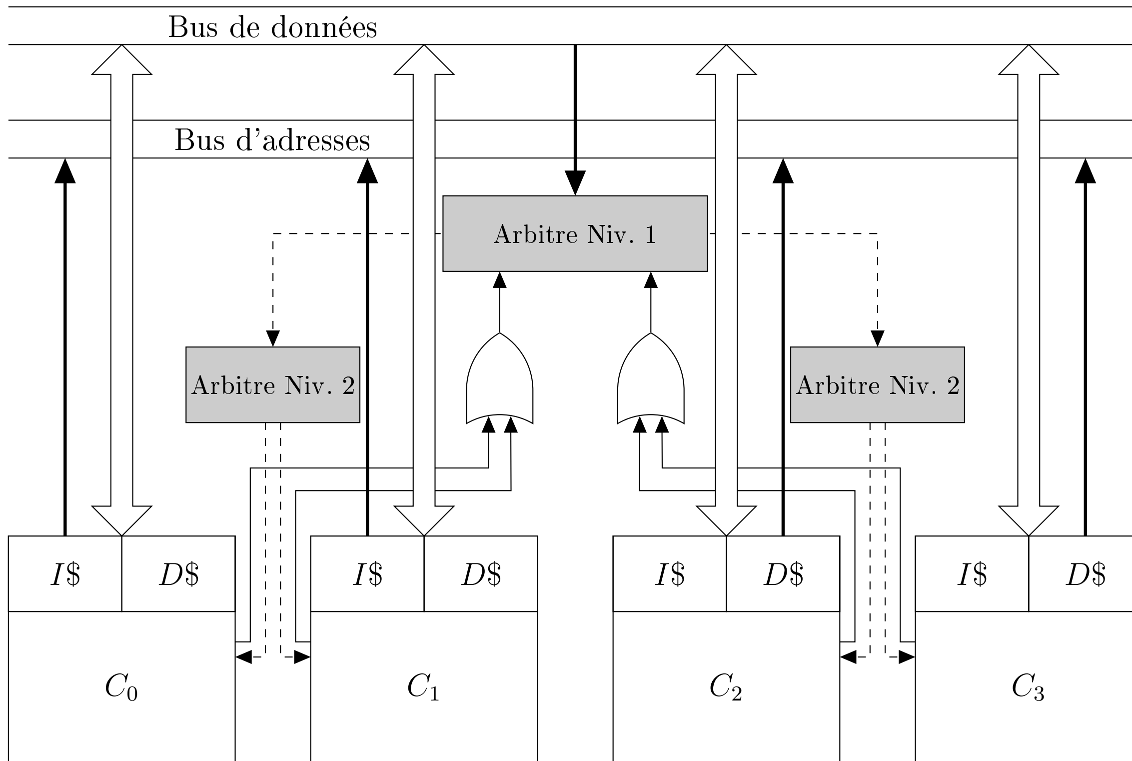

FIGURE 3.1 – Représentation schématique de l'architecture multi-cœurs cible. L'interconnexion est assurée par un bus partagé administré par un arbitre à deux niveaux.

3.1 Vers un modèle d'arbitrage multi-niveaux prévisible

3.1.1 Un arbitrage de bus à deux niveaux

Afin de répondre à la problématique précédemment énoncée, nous proposons d'introduire un arbitre de bus à deux *niveaux* (ou étages) basé sur un principe de rangement des cœurs dans des *groupes* de priorité différents. Chaque cœur ne peut appartenir qu'à un seul et unique groupe. Au premier niveau d'arbitrage, on alloue un créneau d'occupation du bus à un groupe, puis au second niveau, un cœur appartenant à ce groupe est sélectionné et reçoit l'autorisation d'accéder au bus. La figure 3.1 décrit une plateforme à quatre cœurs interconnectée par un bus partagé dont le fonctionnement est régi par un arbitre à deux niveaux. Dans l'exemple présenté, les cœurs sont répartis dans deux groupes de deux cœurs chacun.

Sur chaque cœur, lorsqu'une donnée ou une instruction requise par la tâche exécutée n'est pas présente en cache, le contrôleur de bus du cœur envoie une *requête* d'accès directement à l'arbitre de premier niveau. Ce signal *request* est représenté sur la figure par une ligne pleine. Les requêtes des cœurs appartenant au même groupe aboutissent à la même entrée de l'arbitre. L'arbitre traite ensuite les requêtes des différents groupes suivant deux politiques que nous détaillons en sections 3.2 et 3.3. Le signal *grant*, par l'intermédiaire duquel l'arbitre de premier niveau *accorde* un slot d'accès au bus à l'un ou l'autre des groupes, est d'abord envoyé à l'arbitre de second niveau dont dépend le

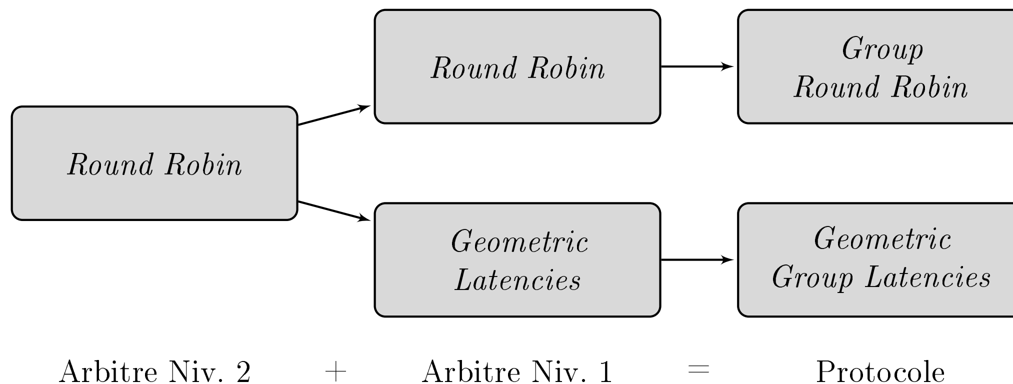

FIGURE 3.2 – Les deux mécanismes d'arbitrage proposés.

cœur qui a émis la requête. L'arbitre de second niveau renvoie ensuite ce signal au cœur qui en a fait la demande. Si plusieurs cœurs du même groupe sont en attente d'un slot d'accès au bus, la sélection du cœur à servir se fait en suivant la politique du tourniquet. Finalement, le cœur sélectionné effectue l'opération de lecture ou d'écriture comme sur une plateforme classique à un seul arbitre (cf. section 2.2.2).

3.1.2 Deux politiques d'ordonnancement des accès

Le principe sous-jacent du mécanisme présenté ici est que les règles d'arbitrage suivies à chacun des deux niveaux déterminent les latences de bus supportées par les cœurs. Le but recherché est d'ordonnancer des charges de travail hétérogènes sur des systèmes dont le respect des échéances temporelles est critique. C'est pourquoi nous avons besoin d'utiliser à chaque niveau d'arbitrage une politique compatible avec le calcul de temps d'exécution pire-cas, comme celle du tourniquet (ou *Round Robin*). Elle permet d'établir une latence d'accès en fonction du nombre de cœurs et de la latence d'un accès au bus. Ce délai est prévisible et ne dépend pas de la nature des tâches exécutées sur les autres cœurs.

Nous introduisons ensuite l'algorithme *Geometric Latencies* (GL), présenté en section 3.3.1. A la différence de *Round Robin*, les niveaux de bande passante attribués par cet algorithme peuvent être différents suivant les éléments servis. Les latences pire-cas qui résultent du délai entre l'émission d'une requête et le moment où elle est servie, sont proportionnelles à des puissances de deux. En combinant ces deux algorithmes aux deux niveaux d'arbitrage de bus, il serait possible de construire quatre protocoles d'arbitrage différents. Cependant, les formules montrent que *Geometric Latencies* se comporte de la même manière que Round Robin avec un nombre limité de cœurs. Pour cette raison nous n'avons retenu que *Round Robin* pour le second niveau d'arbitrage, c'est-à-dire entre les différents cœurs alloués à un groupe. En revanche nous utilisons à la fois *Round Robin* et *Geometric Latencies* afin d'assurer le premier niveau d'arbitrage permettant la sélection du groupe à servir. Cette combinaison nous permet d'obtenir deux arbitres, comme le montre la figure 3.2.

Ces deux mécanismes doivent être configurés de la même manière : on détermine au préalable le nombre de groupes et le nombre de cœurs à l'intérieur de chacun de ces

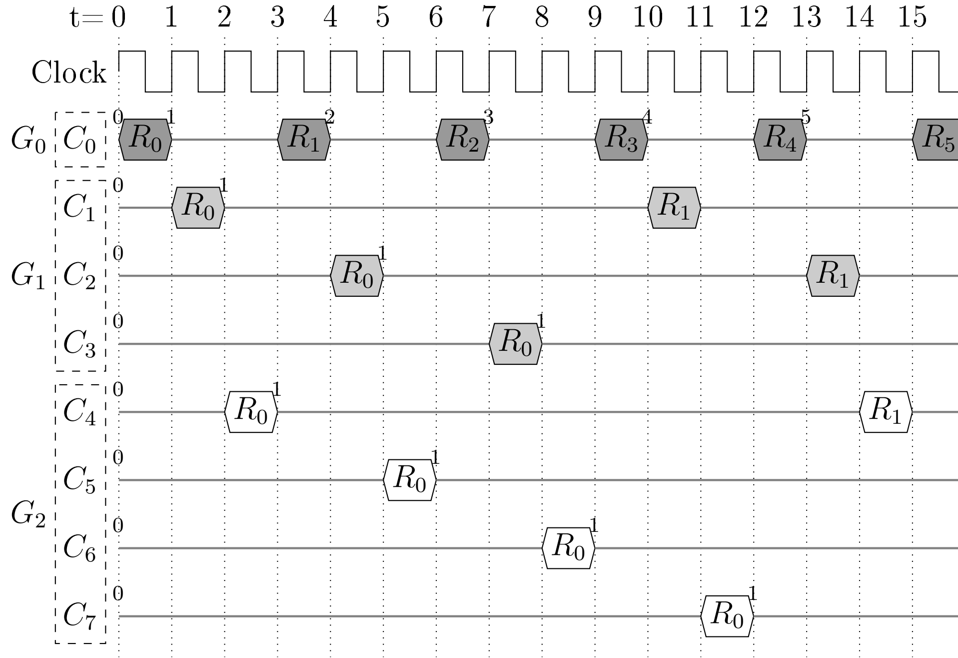

FIGURE 3.3 – Exemple d'arbitrage de bus avec le protocole *Group Round Robin* (GRR-{1,3,4}).

groupes. Soit L_{G_i} la latence pire-cas appliquée au groupe G_i, N_i le nombre de cœurs dans G_i et N le nombre total de groupes. Si les cœurs à l'intérieur d'un groupe sont arbitrés avec *Round Robin*, la latence pire-cas d'accès au bus (*Upper Bound Delay*) appliquée à chacun de ces cœurs est égale à :

$$\forall i \in [0..N-1], \forall j \in [0..N_i-1], L_{C_j \in G_i} = N_i \times L_{G_i} \tag{3.1}$$

La valeur de la latence de groupe L_{G_i} est déterminée à la fois par le nombre total de groupes de la plateforme et par les règles d'arbitrage utilisées par l'arbitre de premier niveau.

3.2 Le protocole *Group Round Robin*

La première des deux politiques d'arbitrage que nous proposons a été nommée *Group Round Robin* (GRR). Elle repose sur l'utilisation de la politique du tourniquet à chacun des deux niveaux d'arbitrage. La figure 3.3 illustre un exemple d'arbitrage de bus partagé en utilisant ce protocole sur une plateforme à huit cœurs notés de C_0 à C_7 répartis sur trois différents groupes notés de G_0 à G_2. La latence d'accès au bus est de un cycle. On considère ici le pire scénario d'ordonnancement possible, où tous les cœurs envoient des requêtes au bus en permanence et selon une politique *in order*, c'est-à-dire qu'une nouvelle requête n'est pas envoyée tant que la requête courante n'est pas satisfaite. La politique du tourniquet est appliquée à chacun des étages de l'arbitre : chaque groupe accède au bus à tour de rôle, c'est-à-dire tous les trois slots dans l'exemple présent, et les cœurs de ce

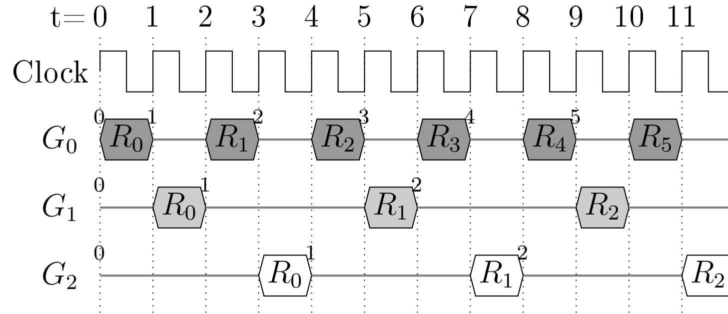

FIGURE 3.4 – Exemple de partage de bus entre trois groupes avec le protocole *Geometric Latencies*.

groupe partagent eux aussi les créneaux attribués à tour de rôle, soit à chaque slot dans le groupe G_0, une fois sur trois pour G_1 et une fois sur quatre pour G_2. Soit L la latence d'accès au bus pour n'importe quelle requête. En utilisant un arbitre fonctionnant suivant GRR avec N groupes et N_i cœurs dans le groupe G_i, la latence pire-cas d'accès au bus appliquée à un groupe est donnée par la formule suivante :

$$\forall i \in [0..N-1], \quad L_{G_i} = N \times L \tag{3.2}$$

A partir de l'équation (3.1), nous en déduisons la latence pire-cas d'accès au bus appliquée à un cœur de ce groupe :

$$\forall i \in [0..N-1], \quad \forall j \in [0..N_i-1], \quad L_{C_j \in G_i} = N_i \times (N \times L) \tag{3.3}$$

3.3 Le protocole *Geometric Group Latencies*

La seconde des deux politiques d'arbitrage que nous proposons a été nommée *Geometric Group Latencies* (GGL). Avec ce protocole, le second niveau d'arbitrage utilise toujours la politique du tourniquet, alors que la sélection des groupes au premier niveau d'arbitrage se fait selon une nouvelle politique intitulée *Geometric Latencies* Dans la section suivante, nous décrivons l'algorithme GL appliqué à un arbitrage à un seul niveau avant d'analyser les latences d'accès au bus produites par *Geometric Group Latencies*.

3.3.1 Présentation de l'algorithme *Geometric Latencies*

Le premier des deux protocoles que nous avons développés, GRR, attribue des latences de bus égales à tous les groupes. La priorisation de l'un ou l'autre des groupes peut uniquement varier en y incluant plus ou moins de cœurs d'exécution. A contrario, dans le protocole *Geometric Latencies* les latences évoluent aussi en proportion inverse de la priorité du groupe, à la manière d'une série géométrique. Dans la suite de cette section, nous notons qu'un groupe est servi pour exprimer le fait qu'à l'intérieur de ce groupe, le cœur sélectionné par l'arbitre de second niveau peut accéder au bus.

Dans l'exemple présenté dans la figure 3.4, qui correspond au pire scénario d'arbitrage, le groupe G_0 peut obtenir l'accès au bus partagé une fois sur deux au maximum, et les

deux autres groupes peuvent y accéder une fois sur quatre au maximum. Si on considère une latence de bus de L cycles, ceci signifie que la première requête à servir du groupe G_0 subit une latence pire-cas de $2.L$ cycles tandis que cette valeur est de $4.L$ pour celles des groupes G_1 et G_2. Il est à noter que les deux derniers groupes subissent toujours la même latence.

3.3.2 Spécification

Afin de décrire la manière dont il serait possible de réaliser une implémentation matérielle du mécanisme d'arbitrage utilisé dans *Geometric Latencies*, nous avons spécifié les signaux régissant le comportement de l'arbitre. Cette modélisation garantit que le scénario pire-cas montré dans la figure 3.4 est celui qui mène toujours aux pires latences de bus pour les cœurs de chaque groupe. Elle doit aussi décrire le comportement de l'arbitre dans tous les autres cas, c'est-à-dire lorsque il n'est pas saturé de requêtes et qu'une hiérarchisation de la priorité des requêtes doit être effectuée.

Le fonctionnement de l'arbitre est régi par les trois signaux de contrôle suivants :

- le signal r_i (*request*) positionné à 1 indique que le groupe G_i a au moins une requête en attente (non servie) ;
- le signal g_i (*grant*) est à 1 lorsque le groupe G_i est servi et est à 0 pour les autres groupes ;
- le signal p_i (*prioritize*) est une variable d'état qui est positionnée à 1 lorsqu'on doit attribuer la priorité au groupe G_i au prochain créneau laissé libre par les groupes d'indice plus faible $G_{j<i}$.

Dans les équations que nous présentons ci-dessous, nous considérons que r_i, g_i et p_i sont des variables booléennes. Le bus est alloué au groupe G_i au cycle t à condition que ce groupe ait une requête en attente (r_i^t), qu'il ait la priorité sur les groupes d'indice plus élevé (p_i^t) et qu'aucun groupe d'indice plus faible n'ait la priorité sur le bus ($\overline{\prod_{j=0}^{i-1} p_j^t}$). Ceci est exprimé par l'équation (3.4) :

$$g_i^t = r_i^t . p_i^t . \prod_{j=0}^{i-1} \overline{p_j^t} \tag{3.4}$$

Nous devons ensuite définir comment se fait l'attribution des priorités entre les différents groupes. Le positionnement du signal p_i^{t+1} à 1 indique que le groupe G_i aura la priorité sur les groupes d'indice plus élevé au cycle $t+1$. Ceci peut se faire dans deux cas précis. Dans le premier cas, G_i a déjà la priorité (sur les groupes d'indice plus élevé $G_{j>i}$) au cycle t mais il ne peut obtenir l'accès au bus immédiatement car un groupe d'indice plus faible a déjà la priorité sur lui, donc cette priorité est reportée sur le cycle suivant :

$$p_i^{t+1} = p_i^t . \sum_{j=0}^{i-1} p_j^t \tag{3.5}$$

Dans le second cas, G_i n'a pas la priorité (sur les groupes d'indice plus élevé $G_{j>i}$) au cycle t mais aucun groupe d'indice plus faible $G_{k<i}$ n'a la priorité sur lui, donc il gagne cette priorité pour le cycle suivant :

$$p_i^{t+1} = \overline{p_i^t} . \prod_{j=0}^{i-1} \overline{p_j^t} \tag{3.6}$$

Ces deux formules peuvent être regroupées en utilisant un *ou exclusif*. Enfin, comme les deux derniers groupes ont le même niveau de priorité, le dernier groupe G_{N-1} a la priorité lorsque l'avant-dernier groupe G_{N-2} ne l'a pas, et inversement. Tout ceci s'exprime par l'équation (3.7) :

$$\begin{cases} p_i^{t+1} & = & p_i^t \oplus \prod_{j=0}^{i-1} \overline{p_j^t} \qquad \forall i \neq N-1 \\ \\ p_{N-1}^{t+1} & = & \overline{p_{N-2}^{t+1}} \end{cases} \tag{3.7}$$

On attribue les créneaux de bus au groupe G_{N-1} avec la même fréquence que pour G_{N-2}. Finalement, nous obtenons l'expression de la latence pire-cas d'accès au bus d'un cœur :

$$\begin{cases} L_{G_i} & = & 2^{i+1} \times L \qquad \forall i < N-1 \\ \\ L_{G_{N-1}} & = & 2^{N-1} \times L \end{cases} \tag{3.8}$$

La preuve de l'équation (3.8) est donnée dans la section suivante.

3.3.3 Preuve de la spécification

Soit δ_i la plus petite distance entre deux créneaux avec $\prod_{j=0}^{i-1} \overline{p_j} = 1$.

Supposons que $g_i^t = 1$. D'après l'équation (3.4), cela signifie que $p_i^t = 1$ et $\prod_{j=0}^{i-1} \overline{p_j^t} = 1$.

D'après l'équation (3.7), nous obtenons $p_i^{t+1} = 0$. De plus $\forall d \mid 1 \leq d < \delta_i$, $\prod_{j=0}^{i-1} \overline{p_j^{t+d}} = 0$, ce qui nous donne $p_i^{t+d+1} = 0$.

Maintenant, $\prod_{j=0}^{i-1} \overline{p_j^{t+\delta_i}} = 1$. Puisque $p_i^{t+\delta_i} = 0$, nous obtenons $g_i^{t+\delta_i} = 0$ (d'après l'équation (3.4)).

De plus, d'après l'équation (3.7), nous savons que $\forall d \mid \delta_i + 1 \leq d < 2\delta_i$, $\prod_{j=0}^{i-1} \overline{p_j^{t+d}} = 0$ et donc que $p_i^{t+d+1} = 1$.

Par conséquent, $g_i^{t+2\delta_i} = 1$. Ceci prouve que la distance la plus courte entre deux créneaux avec $g_i = 1$ est égale à $2\delta_i$.

De plus, $\forall d \mid 2\delta_i \leq d < 3\delta_i$, $\prod_{j=0}^{i-1} \overline{p_j^{t+d}} = 0$ et $p_i^{t+d+1} = 0$.

Ensuite, la distance entre deux créneaux telle que $\prod_{j=0}^{i} \overline{p_j} = 1$ i.e. $\prod_{j=0}^{i-1} \overline{p_j} = 1$ et $p_i = 0$, également notée δ_{i+1}, est donnée par $3\delta_i - \delta_i$: $\delta_{i+1} = 2\delta_i$.

Par conséquent, la distance la plus courte entre deux créneaux avec $g_{i+1} = 1$ est deux fois la distance entre deux créneaux avec $g_i = 1$.

Finalement, les équations (3.4) et (3.7) pour G_0 peuvent être simplifiées par $g_0^t = r_0^t.p_0^t$ et $p_0^{t+1} = \overline{p_0^t}$. Ceci donne $\delta_0 = 2$.

Par récurrence sur i, nous obtenons $\delta_i = 2^{i+1}$.

3.3.4 Implémentation matérielle

Les équations vues à la section précédente permettent non seulement de fournir une spécification formelle du protocole *Geometric Latencies* mais aussi de donner un aperçu de la manière dont il pourrait être implémenté matériellement à l'aide d'une machine de Mealy.

La figure 3.5 montre un exemple d'automate fini qui pourrait être utilisé pour modéliser le comportement du protocole présenté avec quatre groupes de priorité différents. Dans le cas présent, nous considérons que tous les cœurs émettent continuellement des requêtes, *i.e.* $\forall i \in [0..N-1]$, $r_i = 1$. Les nœuds du graphe contiennent la valeur des variables d'état p_i. Elles sont initialisées à 0 sauf pour le dernier groupe. Les valeurs des arcs représentent les résultats de l'algorithme d'arbitrage à travers les signaux d'attribution du bus g_i. Les valeurs de g_i permettent de vérifier que le groupe G_0 accède au bus une fois sur deux, le groupe G_1 une fois sur quatre, et les deux derniers groupes G_2 et G_3 une fois sur huit.

3.3.5 Généralisation à un protocole à deux niveaux

La figure 3.6 illustre un exemple d'arbitrage de bus partagé par huit cœurs en utilisant le protocole *Geometric Group Latencies*. La configuration retenue est GGL-{1,3,4}, ce qui signifie qu'il y a trois niveaux de priorité : le premier cœur dans G_0, les trois cœurs suivants dans G_1 et les quatre derniers dans G_2. Les cœurs envoient des requêtes au bus en permanence.

Le cœur appartenant à G_0 obtient un créneau sur deux. Le groupe G_1 est servi une fois sur quatre, à l'instar du groupe G_2, et les cœurs à l'intérieur de ces deux groupes se partagent équitablement les slots alloués suivant la politique du tourniquet. La configuration retenue implique qu'un cœur appartenant à G_1 peut accéder au bus une fois sur $3 \times 4 = 12$ au maximum, et une fois sur $4 \times 4 = 16$ pour un cœur appartenant à G_2.

En utilisant un arbitre fonctionnant suivant *Geometric Group Latencies* avec N groupes, la latence pire-cas d'accès au bus pour un groupe est donnée par l'équation (3.8). Les slots

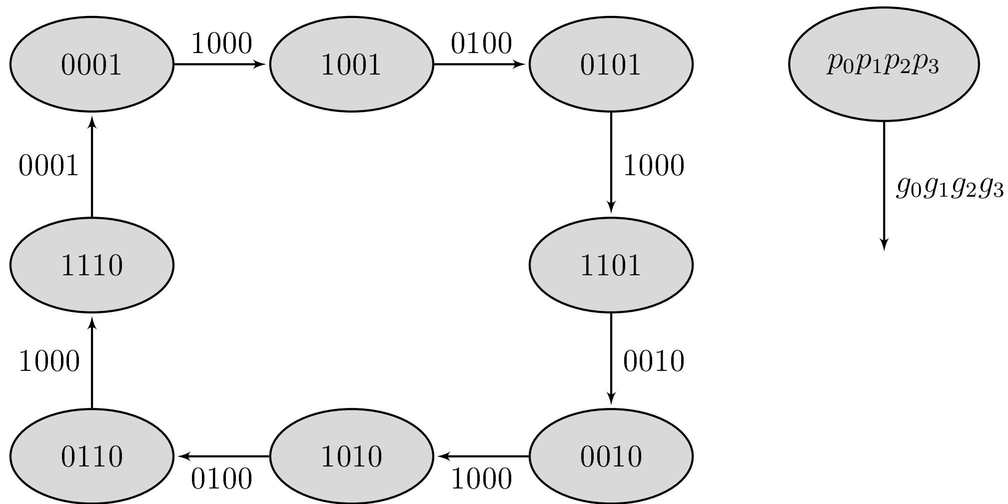

FIGURE 3.5 – Automate fini représentant le comportement d'un arbitre utilisant le protocole *Geometric Latencies*.

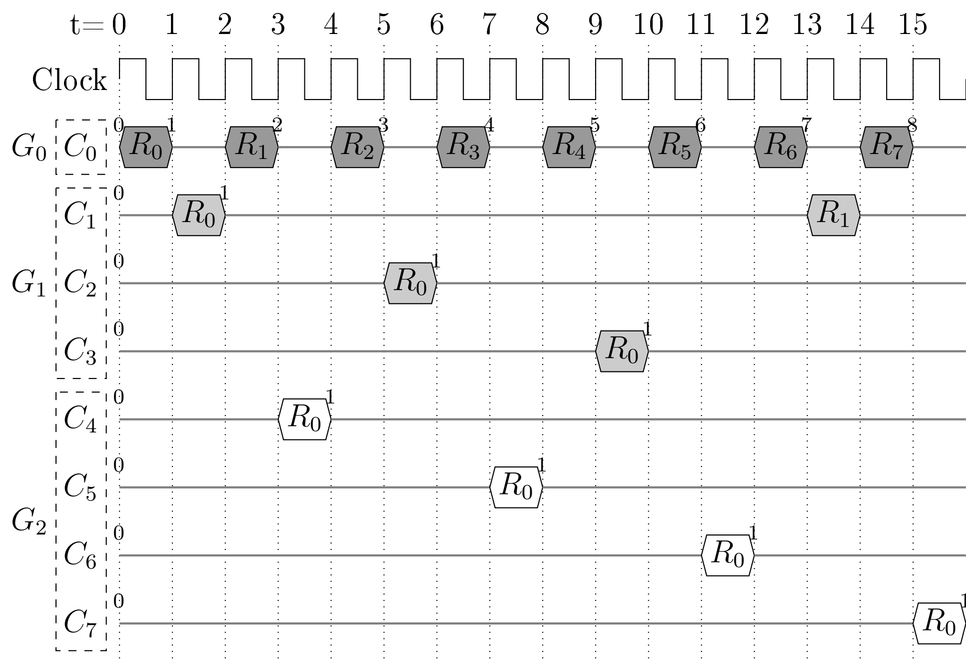

FIGURE 3.6 – Exemple d'arbitrage de bus avec le protocole *Geometric Group Latencies* (GGL-{1,3,4}).

attribués au groupe G_i sont partagés entre les N_i cœurs qu'il inclut suivant la politique *Round Robin*. La latence pire-cas d'accès au bus pour un cœur C_j appartenant au groupe G_i est donc :

$$\begin{cases} L_{C_j \in G_i} & = & (N_i.2^{i+1}) \times L & \forall i \in [0..N-2] \\ L_{C_j \in G_{N-1}} & = & (N_{N-1}.2^{N-1}) \times L \end{cases} \tag{3.9}$$

3.4 Modalités d'évaluation du mécanisme

Nous avons cherché à évaluer l'impact sur la performance pire-cas d'un ensemble de tâches *hétérogènes* de notre mécanisme d'arbitrage à deux niveaux, utilisant une politique adaptée à l'estimation de temps d'exécution par analyse statique. Nous présentons ici le protocole expérimental utilisé pour réaliser nos mesures.

3.4.1 Méthodologie

Outil de mesure

Les protocoles présentés dans ce rapport ont été modélisés à l'aide de la plateforme OTAWA [1] dédiée à l'analyse statique de temps d'exécution pire-cas et qui inclut notamment :

- un *loader* de code binaire supportant plusieurs jeux d'instructions (PowerPC, ARM, TriCore, x86, etc.) ;
- un composant d'analyse de cache d'instructions basé sur des techniques d'interprétation abstraite ;
- un composant d'analyse temporelle qui évalue le temps d'exécution pire-cas des blocs de base, en prenant en compte l'architecture cible et les résultats de l'analyse de cache d'instructions ;
- un *loader* d'informations de contrôle de flot (*flow facts*) capable de lire les annotations fournies par l'outil automatisé oRange [27] ;
- un composant de calcul de temps d'exécution pire-cas qui construit un système ILP (*Integer Linear Programming*) selon la méthode IPET (*Implicit Path Enumeration Technique*). Ce système est résolu en utilisant l'outil `lp_solve` [2].

Jeux de test

Les *benchmarks* utilisés pour ces mesures sont listés dans le tableau 3.1. Les trois premières tâches du jeu de test appartiennent à la collection de *benchmarks* de l'université de Mälardalen, tandis que les autres sont des fonctions du benchmark *Susan* utilisé dans la suite *MiBench*. Certaines de ces fonctions ont été redimensionnées pour obtenir des temps d'exécution suffisants. De plus amples détails sur les caractéristiques des tâches utilisées (temps d'exécution, comportement en mémoire) sont donnés en Annexe A. Puisque nous

1. `http://www.otawa.fr/`
2. `http://lpsolve.sourceforge.net/`

Benchmark	**Function**
`nsichneu`	Simulation d'un réseau de Petri étendu. Code généré automatiquement et contenant beaucoup de structures conditionnelles (plus de 250).
`statemate`	Code généré automatiquement par l'outil STAtechart Real-time-Code generator STARC.
`compress`	Programme de compression de données provenant de la suite de *benchs* SPEC95. La compression est effectuée sur un tampon contenant une petite quantité de données aléatoires.
`susan_corners_quick`	Algorithme de détection d'angles provenant de la suite SUSAN (programme de traitement d'image bas-niveau).
`susan_edges_small`	Algorithme de détection de bords de la suite SUSAN.
`susan_principle`	Algorithme d'application de filtre de la suite SUSAN.
`edge_draw`	Algorithme de tracé de bords de la suite SUSAN.
`corner_draw`	Algorithme de tracé de coins de la suite SUSAN.

TABLE 3.1 – Programmes de test utilisés.

Largeur de l'étage de chargement d'instruction	4
Largeur des étages de décodage, renommage et envoi	2
Taille de la file de chargement	8
Taille de la fenêtre d'instructions	4
Unités fonctionnelles *(latence)*	
UAL entière *(1 cycle)*	1
UAL flottante *(3 cycles)*	1
Multiplieur *(6 cycles)*	1
Diviseur *(15 cycles)*	1
Mémoire *(2 cycles)*	1

TABLE 3.2 – Configuration des cœurs.

voulons analyser les résultats de nos protocoles à deux niveaux indépendamment de l'algorithme utilisé pour ordonnancer les tâches, nous n'utilisons pour le moment qu'un ensemble de 8 tâches, afin que chacune d'entre elles soit attribuée à un cœur différent et unique.

3.4.2 Caractéristiques de l'architecture cible

Les mesures présentées ci-après ont été effectuées en considérant un processeur à 8 cœurs superscalaires à exécution ordonnée implémentant le jeu d'instructions PowerPC et dont les paramètres sont donnés dans le tableau 3.2. Chaque cœur a ses propres caches d'instructions et de données. Les caches d'instructions sont à deux voies, associatifs par ensembles avec des lignes de cache de 16 octets et une capacité totale de 2 Ko. La taille des caches a été délibérément fixée à une valeur faible pour que les résultats soient significatifs étant donnée la faible taille des *benchmarks*. Le remplacement des blocs dans les caches est

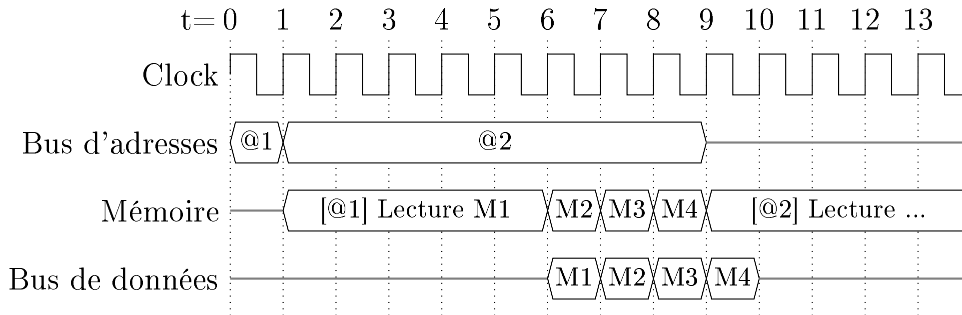

FIGURE 3.7 – Diagramme temporel d'une opération de lecture en mémoire.

assuré par la politique LRU. Enfin, notre outil ne modélisant pas les caches de données, les temps d'exécution obtenus l'ont donc été en considérant d'abord des cœurs ayant un cache de données parfait (*Always Hit*) de la même dimension que le cache d'instructions, puis n'ayant pas de cache de données (*Always Miss*).

La largeur du bus est de 32 bits. Sa latence est d'un cycle, et la latence pire-cas d'une lecture ou d'une écriture en mémoire est de 5 cycles. La figure 3.7 illustre comment est calculée la durée totale du chargement d'une ligne de cache depuis la mémoire. La diffusion de l'adresse cible en mémoire sur le bus d'adresse prend un cycle, la lecture du premier bloc de mémoire (4 octets) prend 5 cycles, la lecture des trois blocs suivants se faisant en un seul cycle pour chacun d'entre eux. Leur transfert sur le bus de données se fait au cycle suivant leur lecture et en parallèle de la lecture du bloc de mémoire suivant. Au final, le chargement d'une ligne de cache depuis la mémoire est effectué en 10 cycles. Mais ceci ne vaut que pour la première itération puisqu'aux chargements suivants, la diffusion de l'adresse cible recouvre la requête précédente : la latence totale de l'opération est alors de 9 cycles. Par conséquent, dans le cas d'un arbitre fonctionnant selon *Round Robin* sur une plateforme à 8 cœurs, la latence mémoire pire-cas appliquée à chaque cœur est de $10 + 7 \times 9 = 73$.

3.4.3 Stratégies d'allocation des tâches aux groupes

Le but des mécanismes d'arbitrage *Group Round Robin* et *Geometric Group Latencies* est de fournir plus de bande passante sur le bus aux cœurs exécutant les tâches les plus critiques, ce qui a pour conséquence de réduire leur temps d'exécution pire-cas. Afin d'améliorer le temps d'exécution global de l'ensemble de tâches de test, il pourrait sembler intéressant d'allouer les tâches les plus longues aux groupes de plus haute priorité. Cependant, considérer uniquement le temps d'exécution pire-cas d'une tâche peut ne pas rendre compte de son comportement vis-à-vis des ressources de stockage partagées et donc du mécanisme d'interconnexion permettant d'y accéder. Une tâche dont l'exécution ne rencontre que peu de défauts de cache accède peu au bus partagé, et donc son temps d'exécution n'est que faiblement affecté par le groupe auquel elle est allouée.

Par la suite, nous proposons d'évaluer l'impact de différents critères d'allocation sur les performances de notre mécanisme :

– le temps d'exécution pire-cas de la tâche sur une plateforme mono-cœur, exprimé en

i	*Benchmark*	WCET 1-cœur	Nb. req. bus	ratio bus/ cache	Sensibilité WCET
Cache de données parfait (AH)					
0	`nsichneu`	529.409	102.767	84,12%	+21,5%
1	`statemate`	786.369	107.788	85,07%	+30,8%
2	`susan_corners_quick`	2.125.304	163.699	53,33%	+71,1%
3	`susan_edges_small`	2.260.287	104.722	23,42%	+43,8%
4	`compress`	671.504	10.107	2,59%	+3,0%
5	`susan_principle`	836.778	2.190	0,79%	+0,9%
6	`edge_draw`	820.343	37	0,01%	+0,0%
7	`corner_draw`	915.205	34	0,01%	+0,0%
Cache de données absent (AM)					
0	`nsichneu`	687.763	206.051	91,39%	+4,5%
1	`statemate`	920.367	177.138	90,49%	+6,1%
2	`susan_corners_quick`	5.016.023	621.595	81,26%	+36,4%
3	`susan_edges_small`	7.186.993	717.400	67,69%	+49,4%
4	`compress`	1.303.706	136.923	26,47%	+5,2%
5	`susan_principle`	5.069.392	440.626	61,60%	+34,8%
6	`edge_draw`	4.274.231	340.048	51,52%	+27,0%
7	`corner_draw`	4.901.337	390.039	50,00%	+30,9%

TABLE 3.3 – Critères des tâches du jeu de test utilisé (taille du cache d'instructions : 2 Ko)

nombre de cycles ;
- le nombre brut de requêtes sur le bus, équivalent au nombre de défauts de cache ;
- le ratio du nombre de requêtes sur le bus par rapport au nombre total de chargements dans le cache (taux de défauts de cache) ;

Les valeurs de ces critères ont été calculées pour chaque tâche du jeu de test, avec cache de données parfait ou sans cache de données et sont regroupées dans le tableau 3.3. Nous introduisons également un nouveau critère, la *sensibilité* du temps d'exécution pire-cas en fonction de la latence de bus.

Sensibilité du temps d'exécution pire-cas d'une tâche à la latence du bus

L'hétérogénéité d'une tâche s'explique par son comportement vis-à-vis de la hiérarchie mémoire : elle génère plus de requêtes mémoire qu'une autre tâche lorsqu'elle rencontre un plus fort taux de défauts de cache ou plus simplement lorsque son temps d'exécution est plus élevé. Nous cherchons à synthétiser ces différences en mesurant leur influence sur le temps d'exécution pire-cas des tâches du jeu de test. La *sensibilité normalisée du temps d'exécution pire-cas* σ_t *à une latence de bus variant de* λ_1 *à* λ_2 d'une tâche t appartenant

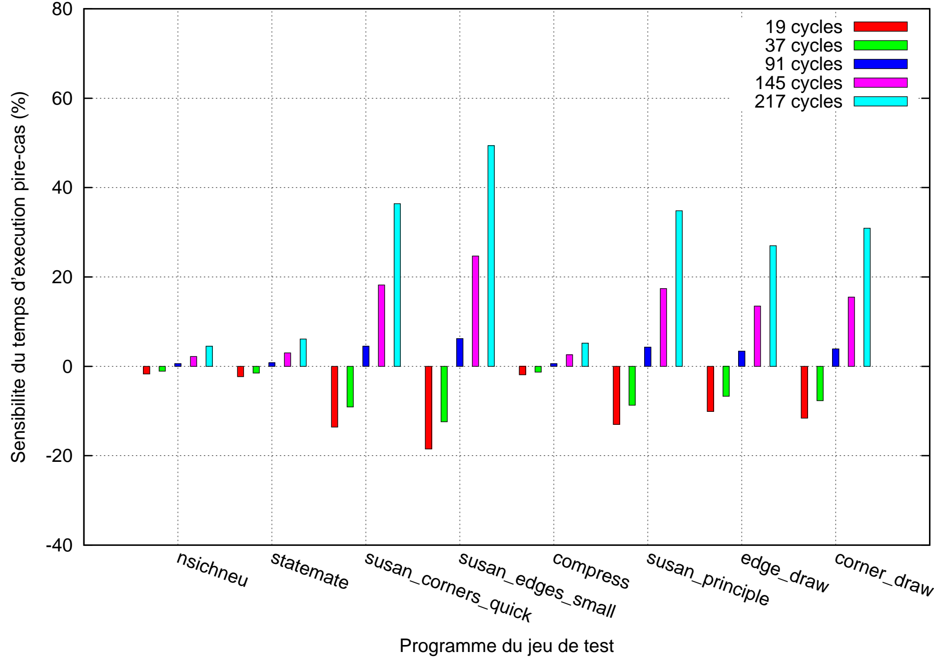

FIGURE 3.8 – Sensibilité du temps d'exécution pire-cas en fonction de la latence de bus et en considérant un cache de données absent (*Data Always-Miss*).

à un ensemble de tâches τ est donnée par la formule suivante :

$$\forall t \in \tau, \sigma_t = \frac{WCET_t^{\lambda_2} - WCET_t^{\lambda_1}}{\sum_{i \in \tau} WCET_i^{\lambda_1}} \tag{3.10}$$

Dans l'équation (3.10), $WCET_t^{\lambda}$ est le temps d'exécution pire-cas de la tâche t en prenant en compte une latence d'accès au bus λ. La sensibilité σ_t est normalisée par rapport à la somme des temps d'exécution pire-cas de toutes les tâches. Ceci rend cette valeur utile afin d'identifier les tâches qui doivent bénéficier d'une meilleure bande-passante en vue d'améliorer la performance globale de l'ensemble de tâches.

Les sensibilités de temps d'exécution pire-cas de chacune des tâches de test sont représentées dans la figure 3.8, avec un cache de données absent (*Always Miss*) et dans la figure 3.9, avec un cache de données *parfait* (*Always Hit*). Elles sont calculées à partir de l'équation (3.10) en prenant comme latence de référence λ_1 la valeur de 73 cycles, qui correspond au délai pire-cas d'accès à un bus arbitré par *Round Robin* sur une plateforme à 8 cœurs. Plusieurs valeurs de latence λ_2 sont utilisées : elles correspondent à des valeurs rencontrées avec les mécanismes d'arbitrage GRR et GGL, comme le montre la table 3.4. A titre d'exemple, avec la configuration GGL-{1,2,5}, le cœur du groupe G_0 subit une latence de 19 cycles, les deux cœurs du groupe G_1 ont une latence de 73 cycles, et ceux du dernier groupe G_2 une latence de 181 cycles. Nous listons les valeurs numériques des sensibilités de WCET dans le tableau 3.3 en utilisant une valeur de λ_2 égale à 217 cycles (cette valeur correspond à la latence maximale pouvant être observée par les cœurs, cf. tableau 3.4).

Les figures 3.8 et 3.9 permettent d'évaluer l'influence de la modification de la latence subie par une tâche sur le temps d'exécution de l'ensemble du jeu de test, c'est-à-dire la

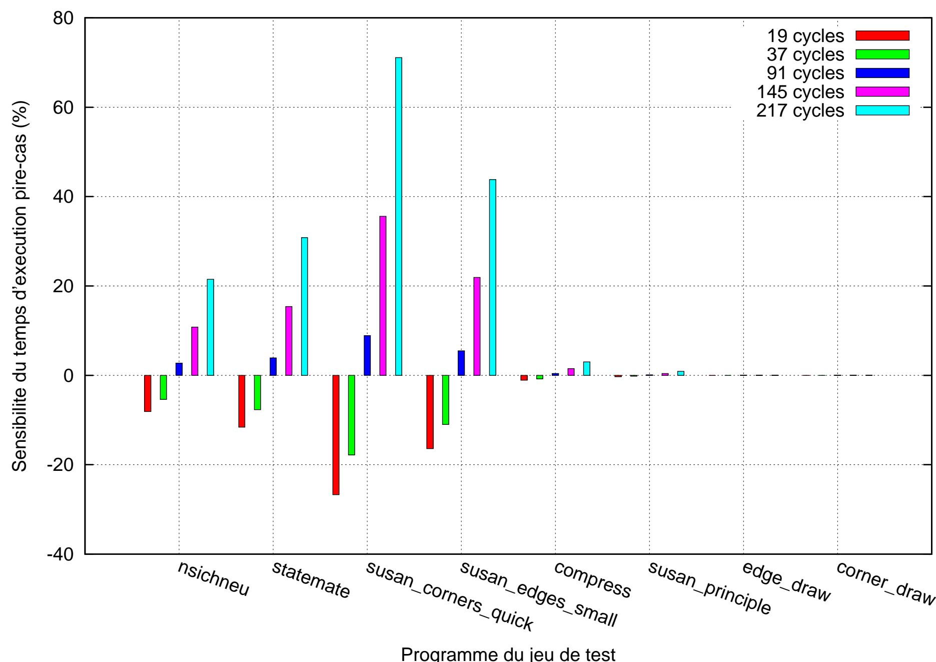

FIGURE 3.9 – Sensibilité du temps d'exécution pire-cas en fonction de la latence de bus et en considérant un cache de données *parfait* (*Data Always-Hit*).

somme des temps d'exécution des tâches qui le composent. Par exemple, avec un cache de données parfait, appliquer une latence pire-cas de 37 cycles à la tâche `susan_edges_small` (par rapport à la latence standard de 73 cycles) améliore le temps d'exécution du jeu de test de 11 %. On observe qu'il est possible d'augmenter la latence d'accès au bus de certaines tâches (par exemple, `compress`) avec un impact modeste sur les performances des tâches, alors qu'une réduction significative du temps d'exécution du jeu de test est réalisable à condition de réduire la latence d'autres tâches, comme `susan_corners_quick`.

Notre approche n'est pas complètement inédite : Mutlu *et al.* [71] ont déjà proposé une approche permettant d'évaluer le *ralentissement dû à la mémoire* (cf. section 2.3.1). Néanmoins, leur protocole est reconfigurable à la volée, et nécessite de mettre à jour les valeurs des facteurs de ralentissement au fur et à mesure de l'exécution des tâches, ce qui n'est compatible ni avec les exigences des systèmes temps-réel stricts ni avec le mode de fonctionnement des outils d'analyse statique. Le calcul de sensibilité du temps d'exécution présenté ici nécessite seulement une analyse *hors-ligne* des tâches à exécuter, et retranscrit un comportement *statistique*, c'est-à-dire basé sur la totalité des opérations mémoire réalisées durant l'exécution des tâches.

3.4.4 Modalités de configuration de l'arbitre

Les arbitres à deux niveaux doivent être configurés pour connaître à la fois le nombre de groupes utilisés et le nombre de cœurs à l'intérieur de chacun de ces groupes. Dans le tableau 3.4, les trois premières colonnes correspondent au nombre de cœurs N_i dans chacun des groupes G_i. Les six colonnes suivantes correspondent aux latences d'accès au bus $L_{C_j \in G_i}$ subies par les cœurs de ces groupes G_i en utilisant le protocole *Group Round*

Configuration			**Latences avec GRR**			**Latences avec GGL**		
N_0	N_1	N_2	$L_{C_j \in G_0}$	$L_{C_j \in G_1}$	$L_{C_j \in G_2}$	$L_{C_j \in G_0}$	$L_{C_j \in G_1}$	$L_{C_j \in G_2}$
8	-	-	***73***	-	-	*73*	-	-
1	7	-	**19**	**127**	-	19	127	-
2	6	-	**37**	**109**	-	37	109	-
3	5	-	**55**	**91**	-	55	91	-
1	1	6	**28**	**28**	**163**	**19**	**37**	**217**
1	2	5	**28**	**55**	**136**	**19**	**73**	**181**
1	3	4	**28**	**82**	**109**	**19**	**109**	**145**
2	1	5	55	28	136	**37**	**37**	**181**
2	2	4	**55**	**55**	**109**	**37**	**73**	**145**
3	1	4	82	28	109	**55**	**37**	**145**
3	2	3	**82**	**55**	**82**	**55**	**73**	**109**
4	1	3	109	28	82	**73**	**37**	**109**
5	1	2	136	28	55	**91**	**37**	**73**

TABLE 3.4 – Configurations de l'arbitre testées.

Robin, puis *Geometric Group Latencies*.

Les valeurs des latences subies par les cœurs dépendent directement de la configuration choisie. Le tableau 3.4 permet de dresser une liste de toutes les configurations possibles, c'est-à-dire toutes les combinaisons possibles de cœurs sur un, deux ou trois groupes. Certaines configurations ne sont pas incluses car elles sont redondantes. Par exemple, un rapide calcul permet de vérifier que la configuration GRR-{6,1,1} est équivalente à GRR-{1,1,6}, et que la configuration GGL-{1,1,6} est équivalente à GGL-{2,6}. De la même manière, GRR-{2,3,3} équivaut à GRR-{3,2,3}, et GGL-{2,3,3} équivaut à GGL-{2,6}. Pour améliorer la lisibilité du tableau, les n-uplets de latences indiqués en gras désignent des configurations non redondantes entre les deux protocoles.

Au final, pour éviter de répéter plusieurs fois des mesures similaires, nous devons tester :

- une configuration à un groupe (correspondant à un arbitrage classique de type *Round Robin*) ;
- trois configurations à deux groupes (les latences sont les mêmes pour GRR et pour GGL) ;
- cinq configurations à trois groupes pour GRR ;
- neuf configurations à trois groupes pour GGL.

Avec un arbitre de bus à deux niveaux utilisant ces 18 configurations différentes, la lecture du tableau montre que les latences de bus peuvent prendre 14 valeurs λ_i différentes échelonnées de 19 à 217 cycles. Pour chacune des tâches du jeu de test et chacune des valeurs de λ_i, nous avons donc effectué un calcul de temps d'exécution pire-cas en considérant :

- soit un cache de données parfait (*Data Always-Hit*) : les succès de cache d'instructions coûtent un cycle, les défauts de cache d'instructions coûtent λ_i cycles, tout accès au cache de données coûte un cycle ;
- soit un cache de données absent (*Data Always-Miss*) : les succès de cache d'instructions coûtent un cycle, les défauts de cache d'instructions coûtent λ_i cycles, tout

accès au cache de données coûte λ_i cycles.

En fonction de ces paramètres, chacune des tâches peut avoir 28 valeurs de temps d'exécution pire-cas différentes. Ce sont donc 224 estimations de temps d'exécution pire-cas qui ont été réalisées au moyen d'un script avant que ces valeurs ne soient combinées pour déterminer les configurations optimales pour chaque protocole.

L'équation (3.11) montre comment il est possible de déterminer pour chaque configuration le nombre de combinaisons $\gamma_{\{N_0,N_1,N_2\}}$ lorsqu'on utilise trois groupes. On calcule d'abord le nombre d'allocations possibles de tâches dans le premier groupe, c'est-à-dire le nombre de combinaisons de N_0 éléments parmi n. Ensuite on calcule le nombre d'allocations possibles dans le second groupe parmi les tâches restantes, c'est-à-dire le nombre de combinaisons de N_1 éléments parmi $n - N_0$. Les tâches restantes vont dans le dernier groupe (une seule combinaison possible) :

$$\gamma_{\{N_0,N_1,N_2\}} = \binom{n}{N_0} \times \binom{n - N_0}{N_1} \tag{3.11}$$

L'équation (3.11) peut être généralisée afin de calculer le nombre de combinaisons $\gamma_{\{N_0,\ldots,N_{N-1}\}}$ sur une plateforme à n cœurs répartis en N groupes comptant N_i cœurs dans chacun d'entre eux :

$$\gamma_{\{N_0,\ldots,N_{N-1}\}} = \prod_{i=0}^{N-2} \binom{n - (\sum_{j=0}^{i-1} N_j)}{N_i} \tag{3.12}$$

La figure 3.10 présente le nombre de possibilités d'allocation de tâches aux groupes pour chaque configuration. En additionnant toutes les combinaisons possibles, 2473 combinaisons différentes ont été évaluées au total.

3.5 Résultats expérimentaux

Nous présentons ici les résultats d'amélioration de temps d'exécution obtenus par les mécanismes d'arbitrage présentés précédemment. Considérant une plateforme à huit cœurs, nous ne retenons que des configurations d'arbitre comprenant trois groupes de priorité différents au maximum.

Dans un premier temps, nous avons cherché à établir toutes les valeurs de performance pouvant être obtenues par notre algorithme, en testant toutes les allocations possibles de tâches et toutes les configurations possibles pour chaque algorithme. Ceci ne peut être réalisé que pour des ensembles réduits de tâches allouées à un petit nombre de groupes, sous peine d'une augmentation drastique de l'espace des solutions à explorer. C'est pourquoi dans un second temps nous cherchons à reproduire les meilleures valeurs des temps d'exécution en utilisant les stratégies d'allocation des tâches aux cœurs vues précédemment. Les résultats obtenus montrent que les arbitres à deux niveaux peuvent substantiellement améliorer les performances d'un ensemble de tâches hétérogènes, à condition que leur allocation soit réalisée avec soin.

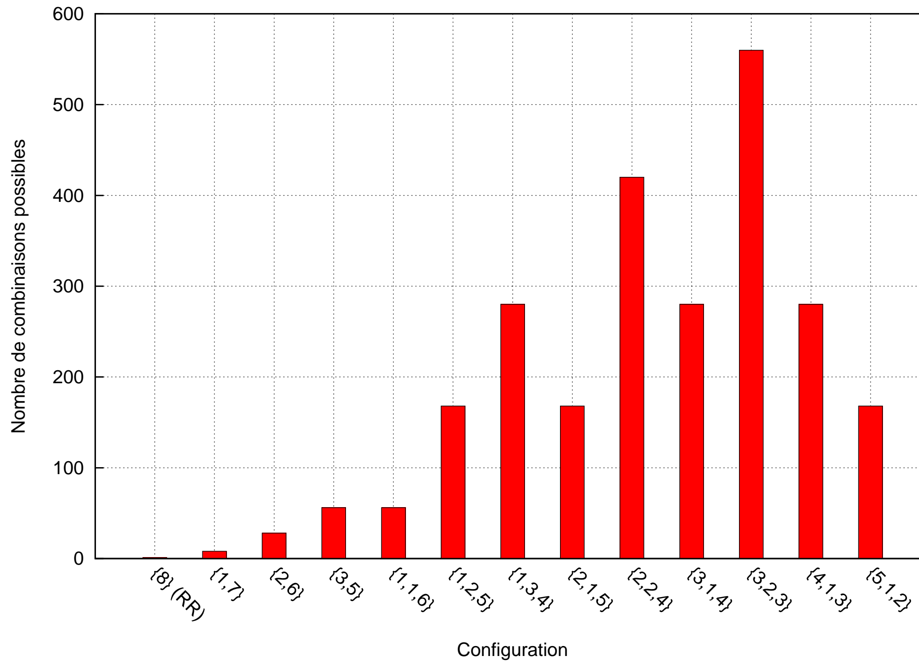

FIGURE 3.10 – Nombre d'allocations possibles de tâches aux groupes $\gamma_{\{N_0,N_1,N_2\}}$ pour chaque configuration $\{N_0, N_1, N_2\}$.

3.5.1 Test exhaustif de toutes les combinaisons

Les tableaux 3.5 et 3.6 présentent les résultats expérimentaux obtenus par notre mécanisme d'arbitrage à deux niveaux. Pour chaque tableau, les trois premières colonnes correspondent au partitionnement des cœurs dans les différents groupes de priorité. Dans le cas de configurations avec trois groupes, *Group Round Robin* et *Geometric Group Latencies* fournissent des structures de latences différentes : une même ligne présente donc des résultats différents suivant que l'on utilise l'un ou l'autre des protocoles. Pour chaque type de cache de données, chaque protocole, chaque configuration des groupes et chaque allocation possible de tâches aux groupes, nous avons calculé deux valeurs différentes : le plus élevé des temps d'exécution pire-cas des tâches exécutées (appelé par la suite temps d'exécution maximum de l'ensemble de tâches ou *WCET Max.*), et la somme des temps d'exécution pire-cas des différentes tâches exécutées (*Somme des WCET*).

La première valeur mesurée peut par exemple être utile pour améliorer le temps d'exécution pire-cas d'une application parallèle dont les *threads* s'exécutant concurremment sont représentés par les tâches du jeu de test : la politique de priorisation suivie par l'arbitre permet d'améliorer le temps d'exécution pire-cas du ou des *threads* critiques, c'est-à-dire ceux provoquant le plus d'attente aux points de synchronisation. La seconde valeur mesurée permet quant à elle d'avoir un aperçu du taux d'*utilisation* des cœurs : plus la somme des temps d'exécution est basse, plus il est possible d'ordonnancer des tâches supplémentaires sur la même plateforme.

Afin de simplifier la présentation des résultats, nous n'avons retenu pour chaque configuration que l'allocation de tâches *optimale* permettant d'obtenir la valeur *minimale* des temps d'exécution. Pour chacun des tableaux, les trois premières colonnes montrent

Configuration			*WCET Max.*		*Somme des WCET*	
N_1	N_2	N_3	**GRR**	**GGL**	**GRR**	**GGL**
8	-	-	*46.021.703*	*46.021.703*	*181.588.379*	*181.588.379*
1	7	-	58.496.999	58.496.999	246.514.523	246.514.523
2	6	-	48.134.578	48.134.578	191.861.771	191.861.771
3	5	-	**36.030.829**	36.030.829	170.941.235	170.941.235
1	1	6	**71.810.338**	**95.486.098**	**255.936.614**	**328.264.133**
1	2	5	53.581.234	71.131.639	195.898.091	248.212.769
1	3	4	37.865.309	50.244.323	184.518.725	233.040.281
2	1	5	53.581.234	71.131.639	195.898.091	227.136.803
2	2	4	37.561.765	49.802.341	**166.591.427**	181.418.639
3	1	4	37.865.309	49.802.341	184.518.725	174.759.233
3	2	3	36.296.698	34.808.477	174.423.776	**161.371.013**
4	1	3	37.865.309	**33.738.971**	184.518.725	166.302.383
5	1	2	53.581.234	36.030.829	195.898.091	175.872.605

Table 3.5 – Meilleurs résultats de temps d'exécution pire-cas (en nombre de cycles) sans cache de données (*Data Always-Miss*).

la configuration utilisée, tandis que les colonnes 4 & 6 (respectivement 5 & 7) donnent les temps d'exécution maximum de l'ensemble de tâches et la somme des temps d'exécution des tâches avec la politique d'arbitre GRR (resp. GGL). La première ligne donne (en italique) les temps d'exécution pire-cas maximum (ou les sommes des temps d'exécution pire-cas) obtenus avec un arbitre utilisant *Round Robin*. Ces valeurs nous servent de point de comparaison avec les résultats obtenus par les arbitres à deux niveaux. Pour chaque colonne, les valeurs minimales et maximales obtenues sont indiquées en gras. Elles permettent d'identifier la ou les configurations amenant aux temps d'exécution les plus bas. La différence par rapport aux temps d'exécution de référence peut être très importante.

Résultats sans cache de données

Le tableau 3.5 présente les meilleures mesures de temps d'exécution réalisées en considérant un processeur sur lequel les cœurs sont dépourvus de cache de données. Avec les allocation optimales de tâches pour chaque configuration, le temps d'exécution maximum de l'ensemble de tâches peut varier de -26,7% à +107,5%, tandis que la somme des temps d'exécution peut varier de -11,1% à +80,8% par rapport aux valeurs de référence (indiquées en italique).

Les résultats obtenus par notre arbitre à deux niveaux montrent que l'utilisation de la politique *Geometric Group Latencies* permet d'atteindre de meilleurs temps d'exécution que *Group Round Robin*. A titre d'exemple, l'allocation de tâches optimale sur trois groupes avec la configuration GGL-{4,1,3} permet d'améliorer le temps d'exécution maximum de l'ensemble de tâches de test de 26,7 % par rapport à *Round Robin*, alors que cette valeur n'est réduite que de 21,7 % en utilisant la configuration GRR-{3,5}. En regardant de plus près la manière dont les tâches sont allouées aux groupes, on constate que la tâche `susan_edges_small`, présentant la sensibilité de temps d'exécution la plus élevée,

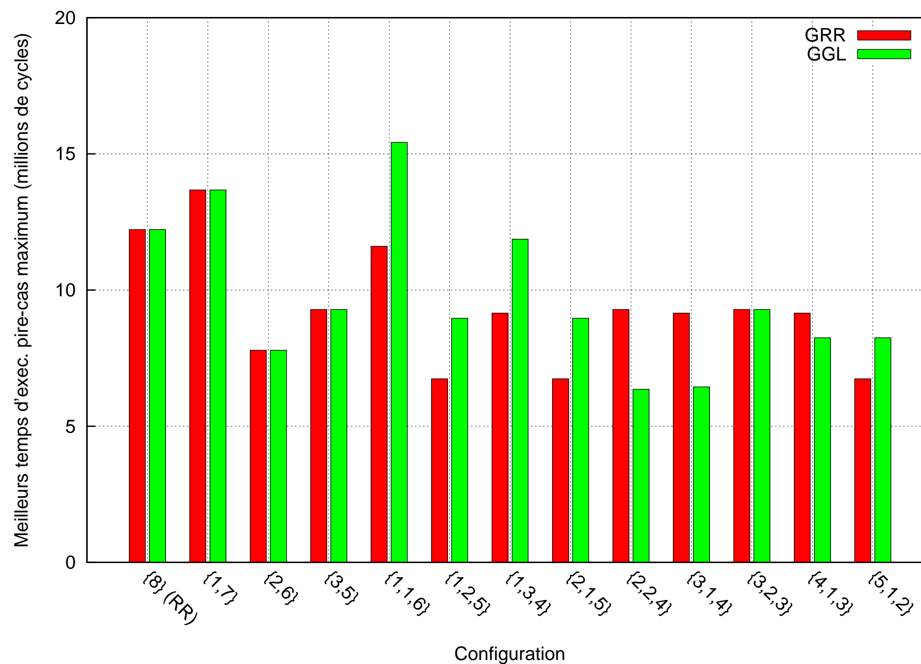

Figure 3.11 – Meilleures sommes des temps d'exécution pire-cas des tâches de test (en nombre de cycles) observés avec un cache de données parfait (*Always Hit*).

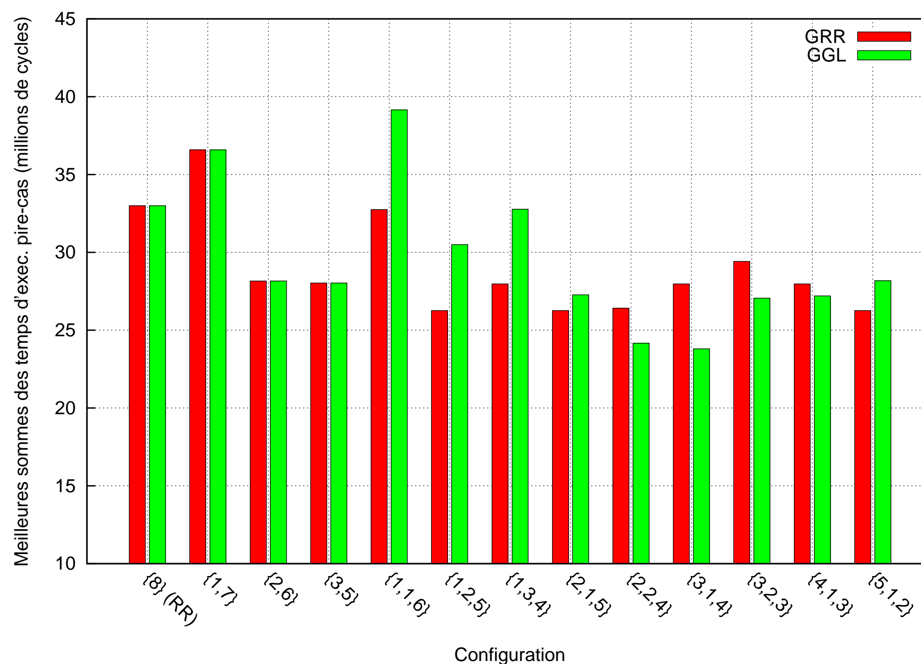

Figure 3.12 – Meilleurs temps d'exécution pire-cas maximum de l'ensemble de tâches de test (en nombre de cycles) observés avec un cache de données parfait (*Always Hit*).

Configuration			*WCET Max.*		*Somme des WCET*	
N_1	N_2	N_3	**GRR**	**GGL**	**GRR**	**GGL**
8	-	-	*12.220.502*	*12.220.502*	*32.994.425*	*32.994.425*
1	7	-	**13.674.729**	13.674.729	**36.586.235**	36.586.235
2	6	-	7.789.634	7.789.634	28.156.061	28.156.061
3	5	-	9.287.276	9.287.276	28.030.691	28.030.691
1	1	6	11.606.462	**15.423.290**	32.752.145	**39.156.455**
1	2	5	**6.743.569**	8.965.039	**26.255.837**	30.494.711
1	3	4	9.154.164	11.866.503	27.968.717	32.778.551
2	1	5	**6.743.569**	8.965.039	**26.255.837**	27.266.933
2	2	4	9.287.276	**6.354.050**	26.414.921	24.157.145
3	1	4	9.154.164	6.441.825	27.968.717	**23.804.507**
3	2	3	9.287.276	9.287.276	29.414.108	27.055.109
4	1	3	9.154.164	8.250.051	27.968.717	27.202.385
5	1	2	**6.743.569**	8.250.051	**26.255.837**	28.177.967

TABLE 3.6 – Meilleurs résultats de temps d'exécution pire-cas (en nombre de cycles) avec un cache de données parfait (*Data Always-Hit*).

est allouée au groupe G_1 garantissant une latence de 37 cycles, alors que les tâches ayant les valeurs de sensibilité les plus faibles (`nsichneu`, `statemate` et `compress`) sont allouées au groupe G_2 avec une latence de 109 cycles. De la même manière, la meilleure somme des temps d'exécution est réduite de 11,1 % en utilisant la combinaison optimale de tâches sur les groupes couplée au protocole GGL-{3,2,3}.

Résultats avec un cache de données parfait

Considérer une architecture de cœur sans cache de données mène évidemment à des résultats pessimistes puisqu'on considère que le bus est constamment sollicité par des requêtes de données, alors qu'en situation d'exécution réelle la bande passante nécessitée est moindre. Comme il a été précisé précédemment, l'outil OTAWA ne permet pas pour le moment une analyse complète du cache de données. C'est pourquoi nous avons également mesuré les temps d'exécution de notre jeu de tâches en considérant un cache de données *parfait* (*Data Always-Hit*). Les résultats obtenus sont présentés dans le tableau 3.6. Avec les allocations optimales de tâches pour chaque configuration, le temps d'exécution maximum de l'ensemble de tâches peut varier de -48,0% à +26,2%, tandis que la somme des temps d'exécution peut varier de -27,9% à +18,7% par rapport aux valeurs de référence (indiquées en italique).

Les meilleurs résultats sont toujours obtenus avec le protocole *Geometric Group Latencies*. La valeur optimale du temps d'exécution maximum de l'ensemble de tâches est réduite de 48,0% avec la configuration GGL-{2,2,4}, alors que celle de la somme des temps d'exécution des tâches est diminuée de 27,9% avec la configuration GGL-{3,1,4}.

Ces améliorations plus importantes des temps d'exécution s'expliquent par le comportement temporel des tâches analysées : si seul le cache d'instructions subit des défauts, la sensibilité des temps d'exécution de certaines tâches est plus élevée, alors que d'autres

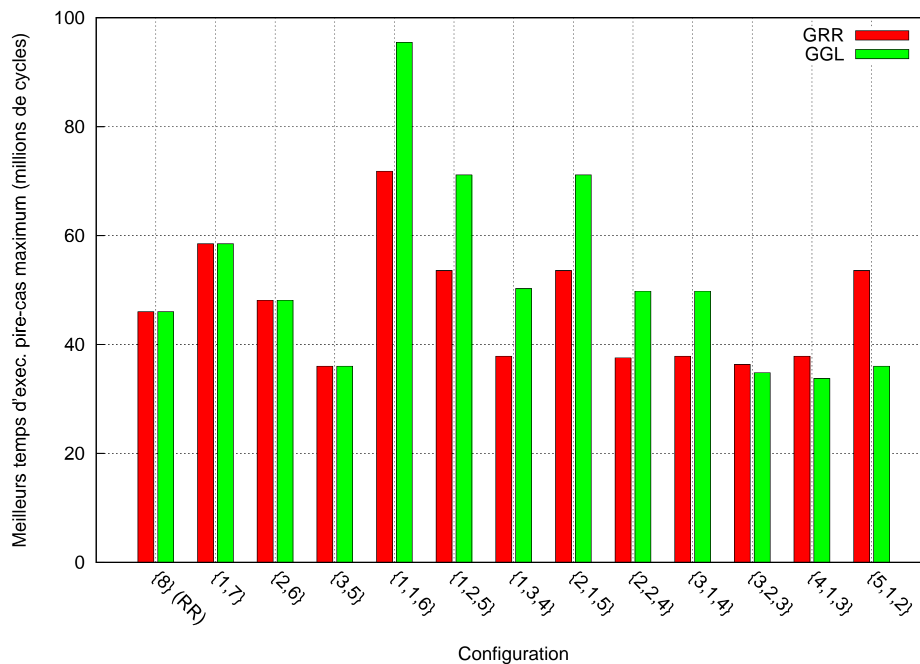

FIGURE 3.13 – Meilleures sommes des temps d'exécution pire-cas des tâches de test (en nombre de cycles) observés sans cache de données (*Always Miss*).

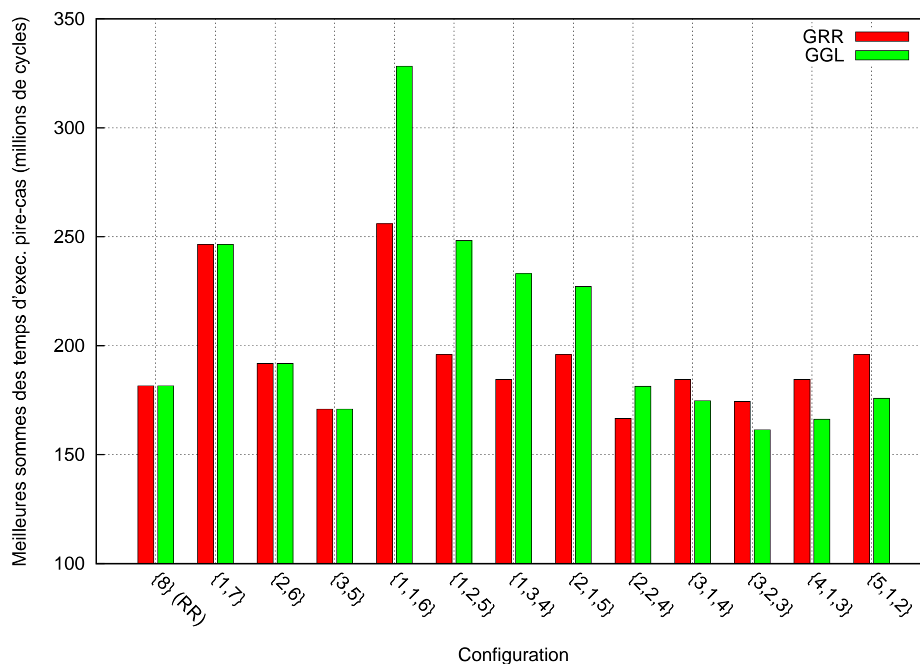

FIGURE 3.14 – Meilleurs temps d'exécution pire-cas maximum de l'ensemble de tâches de test (en nombre de cycles) observés sans cache de données (*Always Miss*).

tâches ne voient leur temps d'exécution que faiblement ou nullement influencé (cf. figure 3.9). Notre mécanisme permet des gains de performances de plus en plus intéressants au fur et à mesure que l'écart-type des valeurs de sensibilités des temps d'exécution augmente.

Les meilleures performances obtenues par *Geometric Group Latencies*, que ce soit avec ou sans cache de données parfait, sont à mettre en relation avec la gestion fine des priorités rendue possible par ce protocole. Ainsi, avec les configurations à trois groupes présentées dans le tableau 3.4, les latences subies par les tâches peuvent varier de 19 à 217 cycles avec le protocole GGL, mais seulement de 28 à 163 cycles pour GRR. GGL permet en outre une plus grande flexibilité en autorisant plus de configurations différentes (13 contre 9 dans notre exemple à 3 groupes). Ce protocole semble donc plus intéressant pour arbitrer un bus sur une plateforme exécutant un ensemble de tâches fortement hétérogènes.

3.5.2 Validation des stratégies optimales d'allocation

Nous cherchons maintenant à déterminer quelle stratégie d'allocation parmi celles présentées en section 3.4.3 permet de s'approcher le plus possible voire d'égaler les performances exhaustives de l'arbitre à deux niveaux vues dans la section précédente. Nous cherchons à établir une hiérarchie préalable de priorités entre les différentes tâches pour déterminer à l'avance celles qui bénéficieraient le plus d'une latence d'accès au bus réduite.

Nous avons retenu quatre critères d'allocation des tâches permettant de rendre compte de leur comportement statistique en mémoire. Un classement décroissant des valeurs de chaque critère étudié est réalisé, comme le montrent les tableaux 3.7 et 3.9 : la priorité des tâches est proportionnelle à ces valeurs. Par conséquent, les tâches les plus longues, bénéficiant d'une sensibilité de temps d'exécution, d'un nombre de requêtes sur le bus ou d'un taux de défauts de cache plus élevés sont allouées au groupe de priorité la plus importante. Ensuite et comme dans la section précédente, nous testons les 18 configurations possibles de notre arbitre à deux niveaux. Les tâches les plus prioritaires suivant le critère retenu sont allouées au groupe de cœurs présentant les latences les plus faibles.

Les tableaux 3.8 (respectivement 3.10) montrent les résultats obtenus avec un cache de données absent (resp. parfait). Pour chaque allocation retenue sur chacune des 18 configurations testées, nous avons calculé les deux valeurs déjà utilisées en section 3.5.1 : le temps d'exécution pire-cas maximum de l'ensemble des tâches du jeu de test (*WCET Max.*), et la somme des temps d'exécution pire-cas des différentes tâches du jeu de test (*Somme des WCET*). La troisième colonne montre la configuration de cœurs dans les groupes ayant conduit au meilleur résultat (présenté en quatrième colonne). Enfin, la dernière colonne présente l'écart entre les meilleurs résultats obtenus ici et les résultats *optimaux* constatés en testant toutes les combinaisons possibles de cœurs aux groupes.

Résultats sans cache de données

Nous évaluons les quatre critères d'allocation listés ci-dessus. Tout accès au cache de données est ici comptabilisé comme étant une requête au bus.

Les trois premiers critères retenus (temps d'exécution pire-cas, sensibilité du temps d'exécution, nombre de requêtes au bus) génèrent des allocations de tâches aux groupes similaires. Les combinaisons générées permettent de retrouver exactement les valeurs optimales des temps d'exécution présentées dans le tableau 3.5. En ce qui concerne le meilleur

i	***Benchmark***	**WCET brut**
3	`susan_edges_small`	7.186.993
5	`susan_principle`	5.069.392
2	`susan_corners_quick`	5.016.023
7	`corner_draw`	4.901.337
6	`edge_draw`	4.274.231
4	`compress`	1.303.706
1	`statemate`	920.367
0	`nsichneu`	687.763

i	***Benchmark***	**Sensib. WCET**
3	`susan_edges_small`	+49,4%
2	`susan_corners_quick`	+36,4%
5	`susan_principle`	+34,8%
7	`corner_draw`	+30,9%
6	`edge_draw`	+27,0%
1	`statemate`	+6,1%
4	`compress`	+5,2%
0	`nsichneu`	+4,5%

i	***Benchmark***	**Nb. req. bus**
3	`susan_edges_small`	717.400
2	`susan_corners_quick`	621.595
5	`susan_principle`	440.626
7	`corner_draw`	390.039
6	`edge_draw`	340.048
0	`nsichneu`	206.051
1	`statemate`	177.138
4	`compress`	136.923

i	***Benchmark***	**ratio bus/ cache**
0	`nsichneu`	91,39%
1	`statemate`	90,49%
2	`susan_corners_quick`	81,26%
3	`susan_edges_small`	67,69%
5	`susan_principle`	61,60%
6	`edge_draw`	51,52%
7	`corner_draw`	50,00%
4	`compress`	26,47%

TABLE 3.7 – Priorisation des tâches en fonction des différents critères retenus avec un cache de données absent.

Critère d'allocation	**Valeur mesurée**	**Meilleure configuration**	**Meilleur résultat**	**Ecart vs. optimum**
WCET brut	*WCET Max.*	GGL-{4,1,3}	33.738.971	=
	Somme des WCET	GGL-{3,2,3}	161.371.013	=
Variab. WCET	*WCET Max.*	GGL-{4,1,3}	33.738.971	=
	Somme des WCET	GGL-{3,2,3}	161.371.013	=
Nb. req. bus	*WCET Max.*	GGL-{4,1,3}	33.738.971	=
	Somme des WCET	GGL-{3,2,3}	161.371.013	=
ratio bus/cache	*WCET Max.*	RR-{8}	46.021.703	+36,4%
	Somme des WCET	RR-{8}	181.588.379	+12,5%

TABLE 3.8 – Temps d'exécution obtenus en fonction des différents critères d'allocation des tâches aux groupes avec un cache de données absent.

i	***Benchmark***	**WCET brut**
3	susan_edges_small	2.260.287
2	susan_corners_quick	2.125.304
7	corner_draw	915.205
5	susan_principle	836.778
6	edge_draw	820.343
1	statemate	786.369
4	compress	671.504
0	nsichneu	529.409

i	***Benchmark***	**Sensib. WCET**
2	susan_corners_quick	+71,1%
3	susan_edges_small	+43,8%
1	statemate	+30,8%
0	nsichneu	+21,5%
4	compress	+3,0%
5	susan_principle	+0,9%
7	corner_draw	+0,0%
6	edge_draw	+0,0%

i	***Benchmark***	**Nb. req. bus**
2	susan_corners_quick	163.699
1	statemate	107.788
3	susan_edges_small	104.722
0	nsichneu	102.767
4	compress	10.107
5	susan_principle	2.190
6	edge_draw	37
7	corner_draw	34

i	***Benchmark***	**ratio bus/ cache**
1	statemate	85,07%
0	nsichneu	84,12%
2	susan_corners_quick	53,33%
3	susan_edges_small	23,42%
4	compress	2,59%
5	susan_principle	0,79%
7	corner_draw	0,01%
6	edge_draw	0,01%

TABLE 3.9 – Priorisation des tâches en fonction des différents critères retenus avec un cache de données parfait.

temps d'exécution, c'est la configuration GGL-{4,1,3} qui permet d'obtenir les meilleurs résultats. La tâche `susan_edges_small`, ayant le temps d'exécution et la sensibilité de temps d'exécution les plus élevés, est allouée au groupe G_1 qui garantit une latence de bus de 37 cycles. Les tâches `susan_corners_quick`, `susan_principle`, `corner_draw` et `edge_draw` sont allouées au groupe G_0 ayant une latence de 73 cycles, et les trois tâches restantes vont dans le dernier groupe.

Pour la somme des temps d'exécution, c'est la combinaison GGL-{3,2,3} qui mène aux meilleures valeurs. Les trois tâches de plus haute priorité (`susan_edges_small`, `susan_principle` et `susan_corners_quick`) sont rangées dans le groupe G_0, les deux suivantes (`corner_draw` et `edge_draw`) vont dans le groupe G_1 et les trois tâches restantes vont dans le dernier groupe.

Résultats avec un cache de données parfait

Les résultats obtenus en considérant des caches de données parfaits montrent que les quatre critères retenus ici ne produisent pas des allocations de tâches aux cœurs identiques. A titre d'exemple, le temps d'exécution maximum du jeu de tâches peut être supérieur de 46,2% à celui obtenu en testant toutes les combinaisons possibles (il reste cependant bien inférieur à la valeur obtenue avec un arbitre utilisant *Round Robin*). De la même manière, la somme des temps d'exécution varie elle aussi de +0% à +18,3% par rapport aux valeurs

Critère d'allocation	**Valeur mesurée**	**Meilleure configuration**	**Meilleur résultat**	**Ecart vs. optimum**
WCET brut	*WCET Max.*	GRR-{2,6}	7.789.634	+22,6%
	Somme des WCET	GRR-{2,6}	28.156.061	+18,3%
Variab. WCET	*WCET Max.*	GGL-{2,2,4}	6.354.050	=
	Somme des WCET	GGL-{3,1,4}	23.804.507	=
Nb. req. bus	*WCET Max.*	GGL-{3,1,4}	6.441.825	+1,4%
	Somme des WCET	GGL-{3,1,4}	23.804.507	=
ratio bus/cache	*WCET Max.*	GRR-{2,2,4}	9.287.276	+46,2%
	Somme des WCET	GGL-{3,1,4}	25.465.457	+7,0%

TABLE 3.10 – Temps d'exécution obtenus en fonction des différents critères d'allocation des tâches aux groupes avec un cache de données parfait.

optimales. La meilleure allocation est réalisée en utilisant le critère de la sensibilité : elle est celle qui permet d'atteindre les valeurs *optimales* des temps d'exécution (cf. section 3.5.1).

En ce qui concerne les mesures réalisées sans cache de données, trois des quatres stratégies testées produisent des allocations similaires et des performances identiques pour la meilleure configuration testée. On observe notamment un forte corrélation entre le temps d'exécution et la valeur de sensibilité : en l'absence d'un cache de données, les tâches les plus longues sont aussi celles qui nécessitent la plus grosse fraction de la bande passante sur le bus. Il s'agit visiblement d'une caractéristique particulière des tâches utilisées dans le jeu de test, puisqu'une tâche longue peut aussi effectuer peu d'accès au bus, notamment si elle est composée de boucles ayant un grand nombre d'itérations et tenant en cache, et si elle n'a pas ou peu besoin de charger des données depuis la mémoire.

Pour les deux architectures testées (sans cache de données, et avec des caches de données parfaits), le critère d'allocation le plus pertinent semble être la valeur de *sensibilité* du temps d'exécution pire-cas. Il permet à chaque fois de déterminer précisément quelle(s) tâche(s) bénéficient le plus d'une latence d'accès au bus réduite, afin de réduire le temps d'exécution global du jeu de test et ainsi améliorer l'ordonnançabilité de la plateforme. En outre le calcul de la valeur de sensibilité est relativement peu coûteux, puisqu'il nécessite seulement deux estimations du temps d'exécution de la tâche avec deux latences de bus différentes, et ces valeurs ont déjà dû être calculées auparavant afin de pouvoir utiliser notre arbitre à deux niveaux. Les résultats montrent que les meilleures valeurs de temps d'exécution maximum et de somme des temps d'exécution du jeu de test sont systématiquement identiques lorsque toutes les allocations de cœurs aux groupes sont testées, et lorsqu'une seule allocation est réalisée suivant le critère de la sensibilité du temps d'exécution.

Conclusion

Nous avons présenté dans ce chapitre un nouveau mécanisme d'arbitrage de bus à deux niveaux, permettant une analyse précise des latences mémoire subies par une tâche

exécutée sur une architecture multi-cœurs. Les deux politiques d'ordonnancement des accès au bus utilisées (*Group Round Robin* et *Geometric Group Latencies*) introduisent plusieurs niveaux de priorités, afin de s'adapter à des ensembles de tâches hétérogènes dont les besoins en bande passante sur le bus sont différents. Les cœurs sont attribués à des groupes de priorité garantissant des latences de bus pire-cas différentes, et totalement prévisibles : elles ne dépendent pas du comportement des tâches accédant au bus partagé. La *configuration* de l'arbitre, c'est-à-dire la politique utilisée, le nombre de groupes et le nombre de cœurs dans chaque groupe, détermine à elle seule les latences subies par les cœurs.

La flexibilité offerte par ce mécanisme permet d'améliorer significativement la performance pire-cas de notre plateforme de test. Les résultats montrent que dans le cas le plus défavorable (sans cache de données), le protocole GGL permet d'améliorer le temps d'exécution maximum du jeu de test de 26,7%, et de réduire la somme des temps d'exécution des tâches de 11,1% par rapport à une plateforme utilisant un arbitrage de bus conventionnel (*Round Robin*). En considérant un cache de données parfait, les différences de comportement entre les tâches du jeu de test s'accentuent, ce qui permet d'améliorer ces valeurs respectives de 48,0% et de 27,9%.

L'étendue des tests menés (plus de 2.400 combinaisons de tâches et de configurations évaluées) montre que l'allocation des tâches aux groupes doit être effectuée avec le plus grand soin si l'on veut obtenir une amélioration significative des temps d'exécution. Mais évaluer l'ensemble des possibilités d'allocation n'est pas une perspective raisonnable si le nombre de cœurs ou de groupes vient à augmenter significativement. C'est pourquoi nous introduisons la notion de *sensibilité* du temps d'exécution pire-cas d'une tâche par rapport à la latence de bus subie. Cette valeur permet de mieux identifier les tâches fortement pénalisées par un traitement *équitable* des accès aux ressources de stockage partagées, et qui bénéficieraient le plus d'une augmentation de bande passante. Les mesures effectuées montrent qu'en allouant les tâches ayant la sensibilité de temps d'exécution la plus élevée aux groupes garantissant la latence de bus la plus faible, on obtient les performances optimales pouvant être délivrées par notre mécanisme.

Chapitre 4

Répartition et ordonnancement de charges de travail hétérogènes complexes sur une architecture multi-cœurs

L'idée d'un mécanisme d'arbitrage de bus à deux étages est maintenant étendue à des systèmes à charges de travail *complexes*, où le nombre de tâches ayant des contraintes temps-réel strictes dépasse le nombre de cœurs disponibles. Par conséquent, chacun des cœurs n'est plus dédié à l'exécution exclusive d'une seule tâche mais doit être partagé efficacement entre plusieurs d'entre elles.

Nous nous intéressons ici à l'allocation de tâches aux cœurs sur des systèmes multi-cœurs hétérogènes. Il existe deux types de stratégies d'allocation. Dans les stratégies basées sur le partitionnement en sous-ensembles de tâches, le mécanisme d'allocation assigne toutes les occurrences d'une même tâche au même cœur : il n'y a pas de migration de tâches possible. Dans les stratégies n'utilisant pas le partitionnement, chaque occurrence d'une tâche peut être exécutée sur un cœur différent.

Cependant, l'utilisation des arbitres à deux niveaux sur notre plateforme fait que les temps d'exécution des tâches varient suivant le cœur auquel elles sont allouées. Les tests d'ordonnancement utilisés dans les approches sans partitionnement ne permettent pas de prendre en compte cette variabilité. C'est pourquoi nous considérons ici une approche d'allocation partitionnée. De cette manière, une fois la répartition des tâches entre les différents cœurs effectuée, l'ordonnancement de chaque sous-ensemble de tâches peut être réalisé avec un algorithme classique d'ordonnancement mono-processeur.

Dans ce chapitre, nous proposons un algorithme *hors-ligne* de partitionnement et d'allocation d'ensembles de tâches aux cœurs permettant de tirer parti des performances des arbitres de bus à deux niveaux. Le but recherché est de réduire la charge globale du système, tout en veillant à respecter les contraintes temporelles des tâches exécutées.

4.1 Spécification du problème

De manière générale, un algorithme d'allocation multi-cœurs (ou multi-processeurs) cherche à répartir un ensemble de tâches (charge de travail) τ composé de n tâches indépendantes $(t_0, ..., t_{n-1})$ sur un ensemble de m cœurs $(c_0, ..., c_{m-1})$. Nous utilisons un modèle de tâches périodiques, dans lequel les occurrences d'une même tâche arrivent à intervalles réguliers. Les dates d'arrivée des différentes tâches sont indépendantes les unes des autres.

Dans cette configuration, chaque tâche t_i est caractérisée par son temps d'exécution pire-cas noté C_i, sa période P_i ainsi que par sa date d'échéance D_i. Nous considérons ici des tâches ayant des dates d'échéance *implicites*, *i.e.* $D_i = P_i$. Le taux d'utilisation d'une tâche t_i est noté v_i et est défini par :

$$\forall i \in [0, n-1],\ 0 < v_i = \frac{C_i}{P_i} \leq 1 \tag{4.1}$$

Nous définissons dans ce chapitre un taux d'utilisation *unitaire* u, qui correspond à une valeur de référence de l'utilisation de chaque tâche sur une plateforme multi-cœurs classique arbitrée par *Round-Robin*. Le taux d'utilisation v_i étant proportionnel au temps d'exécution pire-cas C_i, une tâche présentant une sensibilité du temps d'exécution pire-cas à la latence de bus importante (cf. section 3.4.3) verra son taux d'utilisation varier de façon plus importante autour du taux d'utilisation *unitaire* en fonction du niveau de priorité qui lui est accordé par l'arbitre de bus.

Déterminer la meilleure allocation des tâches aux cœurs est un problème NP-difficile [40]. Les solutions qui s'en approchent reposent sur un compromis entre complexité calculatoire et efficacité du partitionnement obtenu [73]. Les travaux existant utilisent généralement des heuristiques dérivées du problème de *bin-packing*, comme [31] ou [14]. Le problème de *bin-packing* est bien adapté au cas de tâches ayant un taux d'utilisation *statique*, c'est-à-dire ne variant pas suivant le cœur auquel elles sont allouées. Ceci est le cas lorsque l'arbitrage de bus se fait suivant *Round-Robin* : la latence de bus et par conséquence le taux d'utilisation des tâches ne peuvent varier qu'en fonction du nombre de cœurs de la plateforme.

La validité de l'allocation de tâches aux cœurs doit être vérifiée à l'aide d'un algorithme d'ordonnancement. Dans le problème présent, l'allocation est partitionnée, ce qui permet d'utiliser un algorithme classique d'ordonnancement mono-processeur pour assurer que chaque sous-ensemble de tâches respecte ses échéances temporelles (cf. section 4.4).

Dans ce chapitre, nous donnons dans un premier temps un exemple de partitionnement d'un ensemble de tâches avec un algorithme de *bin-packing*. Cependant, les tâches exécutées sur des plateformes multi-cœurs hétérogènes présentent des taux d'utilisation *dynamiques*, en raison de la variabilité des temps d'exécution. Nous proposons un algorithme permettant d'allouer des ensembles de tâches dans ce cas de figure. Les solutions validées par l'algorithme sont ordonnançables avec l'algorithme NP-EDF. Finalement, nous comparons les résultats issus des deux méthodes.

4.2 Partitionnement de charges de travail à taux d'utilisation statique

Nous nous intéressons ici à un problème classique (à une dimension) de *bin-packing*, aussi parfois appelé *problème du sac à dos* [48, 55].

Le problème de *bin-packing* cherche à ranger dans un nombre minimum de boîtes (*bins*) une liste L d'objets a_i telle que :

$$L = (a_0, a_1, ..., a_{n-1}) \mid \forall i \in [0, n-1], 0 < a_i \leq 1 \tag{4.2}$$

Ce problème a été largement étudié par la communauté de recherche opérationnelle depuis le début des années 1970. Il a notamment été montré qu'il était NP-complet [40]. On peut très facilement établir une analogie entre la définition de l'équation (4.2) et notre problématique, qui consiste à partitionner un ensemble de tâches $\tau = (t_0, ..., t_{n-1})$ sur un ensemble de m cœurs $(c_0, ..., c_{m-1})$. Dans le cas présent cependant, chaque cœur c_k avec $k \in [0, m-1]$ accepte une charge de travail U_k telle que $0 \leq U_k \leq 1$. Chaque tâche t_i, $i \in [0, n-1]$ est associée à un taux d'utilisation υ_i défini par $0 < \upsilon_i \leq 1$.

Plusieurs heuristiques de résolution sont généralement utilisées [26], notamment :

- *First-Fit Decreasing* : on trie les objets par ordre de taille décroissante puis on range l'objet courant dans la première boîte qui peut le contenir. Ici, ceci revient à allouer la tâche courante t_i au premier cœur c_k vérifiant la contrainte suivante :

$$U_k + \upsilon_i \leq 1 \tag{4.3}$$

- *Best-Fit Decreasing* : on trie les objets par ordre de taille décroissante puis on range chaque objet dans la boîte la plus remplie qui puisse le contenir. Ici, ceci revient à allouer la tâche courante t_i au cœur c_k vérifiant la contrainte suivante :

$$\forall j \neq k, \quad U_j + \upsilon_i \leq U_k + \upsilon_i \leq 1 \tag{4.4}$$

Ces deux algorithmes ne sont pas optimaux mais permettent d'obtenir de bons résultats. Notons Ω le nombre de boîtes qui seraient utilisées dans une solution optimale. A condition que les objets à ranger soient préalablement triés (du plus gros au plus petit), *First-Fit Decreasing* et *Best-Fit Decreasing* n'utilisent jamais plus de $\frac{11}{9}.\Omega + 1$ boîtes [26].

Dans le cas présent, le tri avant allocation des tâches peut être réalisé suivant différents critères, comme le taux d'utilisation υ_i de chaque tâche, ou son temps d'exécution pire-cas C_i. Dans ce dernier cas, la seule heuristique de résolution facilement utilisable est d'allouer la tâche au cœur avec la plus petite somme des temps d'exécution des tâches s'exécutant déjà dessus.

Nous avons évalué le partitionnement d'un ensemble de tâches à taux d'utilisation statique en effectuant le tri des tâches en fonction de leur taux d'utilisation puis de leur temps d'exécution. Les résultats sont présentés en section 4.5.2.

4.3 Partitionnement de charges de travail à taux d'utilisation dynamique

Dans le chapitre précédent nous avons présenté les protocoles d'arbitrage de bus *Group Round Robin* et *Geometric Group Latencies* permettant d'attribuer des latences de bus

pire-cas différentes suivant le niveau de priorité des cœurs. Il n'est pas possible d'utiliser un algorithme de *bin-packing* pour gérer l'allocation des tâches aux cœurs lorsque le bus est géré par un arbitre à deux niveaux. En effet, l'heuristique utilisée dans cet algorithme ne permet pas de prendre en compte la variabilité des durées d'exécution pire-cas et des taux d'utilisation des tâches en fonction du cœur auquel elles sont allouées.

Le partitionnement de charges de travail à taux d'utilisation dynamique a été étudié en recherche opérationnelle comme étant un problème à n tâches et m machines (ici, cœurs) *différentes* : lorsque les tâches à exécuter sont indépendantes et que leur durée dépend de la machine qui leur est allouée, la méthode la plus efficace pour résoudre le problème est de déterminer la solution optimale de sa représentation sous forme de programme linéaire [29, 94]. On fournit à ce programme les durées d'exécution de chaque tâche sur les machines, puis on calcule le découpage de l'ensemble de tâches en morceaux (ici, ensembles de tâches), chacun étant exécuté sur une machine différente. Finalement, les conditions d'ordonnancement sont vérifiées, ce qui permet de tester la validité de la solution obtenue.

4.3.1 Traduction du problème en système de contraintes linéaires

Un problème linéaire entier comprend tout d'abord une fonction objet à optimiser (c'est-à-dire à minimiser ou à maximiser suivant les besoins). Dans le cas présent, le but recherché est de diminuer autant que possible la charge sur les cœurs. Nous définissons donc une fonction objet U à minimiser et étant égale à la somme des taux d'utilisation de chaque cœur U_k :

$$min: \ U = \sum_{k=0}^{m-1} U_k \tag{4.5}$$

Nous considérons une configuration comprenant m cœurs et N groupes de priorité différents (ces valeurs sont connues au préalable). Il est nécessaire d'ajouter un certain nombre de variables dans le système d'équations avant de pouvoir le résoudre.

La variable booléenne p_x permet de définir la politique d'arbitrage de bus x choisie :

$$\forall x \in (GRR, GGL), \begin{cases} p_x = 1 \text{ si le protocole est } x \\ \\ p_x = 0 \text{ sinon} \end{cases} \tag{4.6}$$

La variable booléenne g_{y-z} permet de définir le nombre de cœurs z dans le groupe G_y. Chaque groupe doit comprendre au minimum un cœur. Étant donné qu'il y a N groupes au total, il ne peut donc y avoir que $m-(N-1)$ cœurs au maximum dans chaque groupe. Chaque variable g_{y-z} est positionnée de la façon suivante :

$$\forall y \in [0, N-1], \ \forall z \in [1, m-(N-1)], \begin{cases} g_{y-z} = 1 \text{ si } G_y \text{ contient } z \text{ cœurs} \\ \\ g_{y-z} = 0 \text{ sinon} \end{cases} \tag{4.7}$$

La variable booléenne $c_k g_y$ permet de définir à quel groupe de priorité G_y appartient chaque cœur c_k :

$$\forall y \in [0, N-1],\ \forall k \in [0, m-1],\ \begin{cases} c_k g_y = 1 \text{ si } c_k \text{ appartient à } G_y \\ \\ c_k g_y = 0 \text{ sinon} \end{cases} \tag{4.8}$$

L'allocation d'une tâche t_i à un cœur c_k est précisée par la valeur de la variable $c_k t_i$:

$$\forall i \in [0, n-1],\ \forall k \in [0, m-1],\ \begin{cases} c_k t_i = 1 \text{ si } t_i \text{ est allouée à } c_k \\ \\ c_k t_i = 0 \text{ sinon} \end{cases} \tag{4.9}$$

Notons C_i^{x-y-z} le temps d'exécution d'une tâche t_i avec un arbitrage de bus suivant une politique x et allouée à un groupe G_y contenant z cœurs, et P_i sa date d'échéance. Le taux d'utilisation υ_i^{x-y-z} d'une tâche t_i exécutée avec les paramètres énoncés ci-dessus est donné par la formule suivante :

$$\forall i \in [0, n-1],\ \forall k \in [0, m-1],\ \upsilon_i^{x-y-z} = \frac{C_i^{x-y-z}}{P_i} \tag{4.10}$$

Les différentes valeurs de υ_i^{x-y-z} doivent être calculées au préalable et sont intégrées dans le système de contraintes sous forme de données numériques. Finalement, le taux d'utilisation de chaque cœur s'écrit de la manière suivante :

$$U_k = \sum_{i,x,y,z} p_x . g_{y-z} . c_k g_y . c_k t_i . \upsilon_i^{x-y-z} \tag{4.11}$$

Le problème d'allocation des tâches aux cœurs comprend un certain nombre de contraintes. Premièrement, la plateforme ne peut avoir qu'une seule politique d'arbitrage de bus, ce qui s'écrit de la façon suivante :

$$\sum_{x \in (GRR, GGL)} p_x = 1 \tag{4.12}$$

Ensuite, chaque groupe de priorité G_y ne peut avoir qu'une quantité unique de cœurs z :

$$\forall y \in [0, N-1],\ \sum_{z=1}^{m-(N-1)} g_{y-z} = 1 \tag{4.13}$$

De plus, chaque cœur c_k ne peut appartenir qu'à un seul et unique groupe G_y. Ceci est assuré par la contrainte suivante :

$$\forall k \in [0, m-1],\ \sum_{y=0}^{N-1} c_k g_y = 1 \tag{4.14}$$

Enfin, chaque tâche t_i ne peut être allouée qu'à un seul et unique cœur c_k. Ceci s'écrit de la façon suivante :

$$\forall i \in [0, n-1],\ \sum_{k=0}^{m-1} c_k t_i = 1 \tag{4.15}$$

Nom du solveur	Version	Source	Remarque
lp_solve	5.5.0.15	LGPL	Non commercial
GLPK (GNU Linear Programming Kit)	4.47	GNU	Non commercial
CPLEX	12.4.0.0	IBM(R) ILOG(R)	Commercial

TABLE 4.1 – Solveurs ILP utilisés.

4.3.2 Algorithme de recherche de solution optimale

Le tableau 4.1 liste les différents logiciels évalués afin de trouver une solution valide, c'est-à-dire une allocation des tâches aux cœurs qui soit ordonnançable avec un taux d'utilisation global U qui soit aussi faible que possible. Implémentant plusieurs algorithmes de recherche de solution optimale, comme la méthode du *simplex* ou celle du *point intérieur*, le logiciel CPLEX a montré des performances supérieures aux deux autres outils tant en rapidité d'exécution qu'en pertinence des résultats. C'est donc le solveur qui a été utilisé pour produire les résultats présentés en section 4.5.

En raison de limitations logicielles inhérentes au mécanisme de résolution du problème, le nombre de variables a été réduit par rapport à la spécification complète décrite dans la section précédente. En effet, les solveurs existants ne tolèrent l'expression de la fonction à optimiser que sous une forme $\sum a_i x_i$ (somme de produits entre une valeur numérique et une seule variable).

Une manière de contourner cette limitation est d'externaliser la gestion du protocole et des configurations. Il devient donc nécessaire de définir au préalable la latence d'accès au bus l_k de chaque cœur c_k en fonction du mécanisme d'arbitrage choisi (GRR ou GGL) et de la répartition des cœurs dans les différents groupes de priorité. Le temps d'exécution de chaque tâche attribuée à ce cœur est corrélé à la valeur de cette latence. Nous notons désormais $C_i^{l_k}$ le temps d'exécution d'une tâche t_i attribuée au cœur c_k. Le taux d'utilisation $\upsilon_i^{l_k}$ d'une tâche t_i exécutée sur un cœur de latence l_k est donné par la formule suivante :

$$\forall i \in [0, n-1],\ \forall k \in [0, m-1],\ \upsilon_i^{l_k} = \frac{C_i^{l_k}}{P_i} \tag{4.16}$$

Il suffit donc de lister au préalable toutes les latences de bus possibles (17 valeurs pour $0 \leq m \leq 7$) et de calculer tous les temps d'exécution de toutes les tâches pour toutes ces latences. Les différentes valeurs de $\upsilon_i^{l_k}$ correspondantes sont intégrées dans le système de contraintes sous forme de données numériques. Le taux d'utilisation de chaque cœur s'écrit maintenant sous la forme suivante :

$$U_k = \sum_i c_k t_i . \upsilon_i^{l_k} \tag{4.17}$$

La méthode utilisée afin de trouver des allocations optimales de tâches aux cœurs est décrite exhaustivement dans l'algorithme 4.1. Cet algorithme a été implémenté dans un script en langage *Perl*. Le principe général de l'algorithme est d'enrichir par itérations successives le système de contraintes ILP en y ajoutant les combinaisons de tâches testées comme étant valides ou invalides, jusqu'à ce que les ensembles de tâches alloués à chaque

cœur soient tous ordonnançables. Il est tout à fait possible que l'algorithme puisse ne pas trouver de solution ordonnançable (cf. résultats en section 4.5.4).

Par la suite, le nombre total de cœurs de la plateforme est noté nb_cores et le nombre de cœurs utilisés pour réaliser les essais d'allocations est noté m. L'algorithme d'allocation teste toutes les possibilités d'allocation, en commençant par utiliser un seul cœur, puis deux, jusqu'à arriver aux nb_cores cœurs du processeur. Les indices des cœurs disponibles vont de 0 à $m-1$. Étant donné le nombre de cœurs utilisés m, toutes les configurations possibles γ de cœurs aux groupes des algorithmes GRR et GGL sont évaluées. Le but est de déterminer un couple configuration / allocation de tâches permettant de minimiser le nombre de cœurs nécessaires ou la charge globale du système, afin de pouvoir par exemple exécuter concurremment d'autres tâches n'ayant pas de contraintes temporelles strictes. L'algorithme ne détermine pas directement la meilleure configuration possible de tâches dans les groupes de priorité (ceci se fait par la suite en comparant les résultats pour chaque configuration).

Phase de calcul des taux d'utilisation

La première partie de l'algorithme permet de calculer les différentes valeurs nécessaires à la recherche d'une solution optimale : latences, temps d'exécution et taux d'utilisation des différentes tâches du jeu de test.

Pour chaque tâche t_i de l'ensemble de tâches τ, une première valeur du temps d'exécution pire-cas C_i^L est calculée en considérant la latence de référence L (ligne 4). Celle-ci correspond à un arbitrage de bus conventionnel utilisant *Round-Robin* et a été calculée au préalable en l. 2 par la formule suivante :

$$L = (m-1) \times L_{bus} \tag{4.18}$$

La période (date d'échéance implicite) de la tâche P_i est calculée en fonction de son temps d'exécution *de référence* et du taux d'utilisation *unitaire* appliqué à toutes les tâches, noté u (l. 5).

Une fois que la période P_i est déterminée (celle-ci reste constante), l'algorithme parcourt les différentes latences de bus l possibles avec la configuration courante γ (l. 10). Pour chaque valeur de l, il calcule le temps d'exécution pire-cas C_i^l de la tâche t_i (l. 11). Il établit ensuite le taux d'utilisation U_i^l de la tâche comme le rapport entre ce temps d'exécution et la période de référence P_i calculée précédemment (l. 12). Les différentes valeurs de taux d'utilisation servent à construire la fonction à minimiser qui sera donnée au solveur ILP. Elles sont donc sauvegardées dans le tableau *ulist* afin d'être utilisées par la suite (l. 13).

Enfin, l'algorithme récupère pour chaque cœur c_k la latence l_k qui lui est appliquée (l. 17). Cette valeur est utile pour gérer les combinaisons de tâches interdites (voir ci-dessous).

Phase d'allocation des tâches aux cœurs

La seconde partie de l'algorithme permet d'injecter les valeurs calculées précédemment dans le solveur ILP utilisé (CPLEX) afin de déterminer une solution optimale. Deux tableaux sont initialisés en lignes 21 et 22. Ils servent respectivement à stocker les combinaisons de tâches non ordonnançables (*blacklist*) et celles qui sont valides (*whitelist*).

Algorithme 4.1 Algorithme d'allocation des tâches aux cœurs.

1: **for** m **in** $[1 : nb_cores]$ **do**
2: $L = \text{rr_latency}(m)$;
3: **for each** task t_i **in** τ **do**
4: $C_i^L = \text{compute_wcet}(t_i, L)$;
5: $P_i = C_i^L / u$;
6: **end for**
7: **for each** configuration γ **in** possible_configurations(m) **do**
8: *// Calcul des taux d'utilisation*
9: **for each** task t_i **in** τ **do**
10: **for each** latency l **in** possible_latencies(γ) **do**
11: $C_i^l = \text{compute_wcet}(t_i, l)$;
12: $v_i^l = C_i^l / P_i$;
13: $\text{add_to}(v_i^l, ulist[i][l])$;
14: **end for**
15: **end for**
16: **for each** available core c_k **do**
17: $l_k = \text{get_latency}(k, \gamma)$;
18: **end for**
19: *// Allocation des tâches aux cœurs*
20: $all_cores_scheduled = \text{false}$;
21: $blacklist = \emptyset$;
22: $whitelist = \emptyset$;
23: $nb_iter = 0$;
24: **while** $!(all_cores_scheduled)$ **do**
25: $nb_iter + +$;
26: $\alpha_{tmp} = \text{solve}(\tau, m, ulist, blacklist, whitelist)$;
27: $all_cores_scheduled = \text{true}$;
28: **if** $(\alpha_{tmp} \neq \emptyset)$ **then**
29: *// Vérification des contraintes d'ordonnançabilité*
30: **for each** taskset τ_k **in** α_{tmp} **do**
31: **if** $(\text{schedulable}(\tau_k))$ **then**
32: $\text{add_to}(\tau_k, whitelist[k])$;
33: **else**
34: $\text{add_to}(\tau_k, blacklist[l_k])$;
35: $all_cores_scheduled = \text{false}$;
36: **end if**
37: **end for**
38: **else**
39: *// Aucune solution ordonnançable*
40: break ;
41: **end if**
42: **end while**
43: $\alpha[m][\gamma] = \alpha_{tmp}$;
44: **end for**
45: **end for**

Tant que tous les ensembles de tâches ne sont pas ordonnançables (l. 20), l'algorithme répète le même mode opératoire : il transforme les différents paramètres en un système de contraintes à l'aide desquelles CPLEX cherche une solution (ensembles de tâches pour chaque cœur). En fonction de la validité de ces ensembles, le systèmes de contraintes est enrichi et le solveur appelé à nouveau, jusqu'à obtention d'une solution complètement ordonnançable.

La suite de cette section indique la manière dont le système de contraintes est généré à partir des données d'entrée. Tout d'abord, le programme indique que la fonction objet U est à minimiser et est égale à la somme des taux d'utilisation de chaque cœur U_k :

$$min:\ U = \sum_{k=0}^{m-1} U_k \tag{4.19}$$

Le fonctionnement interne du solveur ne lui permettant d'optimiser que suivant un seul critère à la fois, il n'est pas possible d'équilibrer la charge entre les différents cœurs. Ceci étant, il est possible de limiter statiquement l'utilisation des cœurs. Le taux d'utilisation du cœur k, noté U_k, est égal à la somme des taux d'utilisation $\upsilon_i^{l_k}$ des tâches qui s'exécutent sur ce cœur. Il doit impérativement être inférieur ou égal à 1 :

$$\forall k \in [0:m-1],\ U_k = \sum_{i=0}^{n-1} (\upsilon_i^{l_k}\ .\ c_k t_i) \leq 1 \tag{4.20}$$

En outre, chaque tâche ne peut être allouée qu'à un seul et unique cœur. Ceci s'écrit de la manière suivante :

$$\forall i \in [0:n-1],\ \sum_{k=0}^{m-1} c_k t_i = 1 \tag{4.21}$$

L'appartenance d'un ensemble τ' de n tâches au tableau $whitelist[k]$ signifie que les tests ont validé l'ordonnançabilité de cet ensemble sur le cœur c_k. L'algorithme va donc *verrouiller* cette allocation valide. Il faut donc que les n tâches appartenant à τ' soient obligatoirement exécutées par c_k et qu'en même temps aucune autre tâche ne puisse plus être allouée à ce cœur. Ceci se traduit par les contraintes suivantes :

$$\tau' \in whitelist[k] \Leftrightarrow (\sum_{t_i \in \tau'} c_k t_i = n) \wedge (\sum_{t_j \notin \tau'} c_k t_j = 0) \tag{4.22}$$

L'appartenance d'un ensemble τ'' de n tâches au tableau $blacklist[l]$ signifie que cet ensemble ne peut pas être ordonnancé sur un cœur de latence l. En revanche, il est possible qu'un sous-ensemble τ''' inclus dans τ'' de $n' < n$ tâches soit ordonnançable sur ce même cœur. On ne doit donc proscrire que le seul cas où les n tâches de τ'' sont allouées au même cœur. Ceci se traduit par les contraintes suivantes :

$$\tau'' \in blacklist[l] \Leftrightarrow \forall c_k \mid l_k = l,\ \sum_{t_i \in \tau''} c_k t_i < n \tag{4.23}$$

Les contraintes correspondantes sont ajoutées au système d'équations linéaires. Par la suite nous présentons les résultats des tests d'allocation en considérant deux possibilités :

– en utilisant seulement l'interdiction des ensembles de tâches non ordonnançables (*blacklist*) ;
– en utilisant également le verrouillage des ensembles de tâches valides (*blacklist* + *whitelist*).

L'interdiction des ensembles de tâches non ordonnançables permet de faire converger l'algorithme vers une solution valide sans remettre en cause son optimalité. Le verrouillage des ensembles des tâches valides permet d'obtenir une solution plus rapidement mais celle-ci est en revanche sous-optimale, car l'espace des solutions explorées est plus restreint.

Si le solveur a trouvé une possibilité d'allocation des tâches aux cœurs respectant les contraintes d'utilisation des cœurs et les contraintes d'allocation issues des itérations précédentes, il fournit en retour à l'algorithme une solution α_{tmp} (l. 28). Elle consiste en un ensemble de valeurs $c_i t_k$ exprimant à quel cœur chaque tâche a été allouée :

$$\alpha_{tmp} = \{c_0 t_0, ..., c_0 t_{n-1}, c_1 t_0, ..., c_{m-1} t_{n-1}\} \tag{4.24}$$

L'algorithme regroupe ensuite les tâches en un sous-ensemble de tâches $\tau_k \in \alpha_{tmp}$ en fonction du cœur c_k auquel elles ont été allouées (l. 30), suivant la formule suivante :

$$\forall k \in [0 : m-1],\ \forall i \in [0 : n-1],\ t_i \in \tau_k \Leftrightarrow c_k t_i = 1 \tag{4.25}$$

Un test d'ordonnancement mono-processeur est ensuite lancé (l. 31) pour chaque ensemble τ_k (cf. section 4.4). Ces différents tests vont permettre à l'algorithme d'enrichir la liste des allocations valides et celle des allocations non ordonnançables. Si τ_k respecte les conditions nécessaires et suffisantes d'ordonnancement avec *Earliest Deadline First*, il est placé dans le tableau $whitelist[k]$ (l. 32). Dans le cas contraire il est ajouté au tableau $blacklist[l_k]$ (l. 34).

Lorsque tous les ensembles de tâches sont ordonnançables, la solution obtenue α_{tmp} est écrite dans le tableau de solutions α (l. 43).

Dans le cas où aucune solution valide (i.e. ordonnançable) n'est atteinte, l'emplacement correspondant dans le tableau reste vide. Toutes les configurations possibles pour chaque quantité de cœurs disponibles sont évaluées.

4.4 Ordonnancement des charges de travail

La phase d'allocation décrite en section 4.3.2 permet de regrouper les tâches en sous-ensembles de tâches γ_k, chacun d'entre eux étant exécuté sur un cœur différent tout en assurant que la charge de travail de ce cœur est inférieure ou égale à 1.

L'ordonnancement des sous-ensembles de tâches ainsi créés correspond à la seconde étape qui permet de valider le bon fonctionnement de la plateforme. Il est en effet primordial de vérifier qu'il existe au moins une combinaison d'exécutions des tâches sur chaque processeur permettant à chaque tâche t_i de respecter ses contraintes temporelles, et notamment sa date d'échéance D_i. On rappelle qu'il n'y a pas de migration de tâches possible, c'est-à-dire que les sous-ensembles de tâches sont figés à l'issue de la phase d'allocation et ne peuvent pas évoluer pendant l'exécution. Par conséquent, l'ordonnancement peut être réalisé de manière indépendante pour chaque sous-ensemble de tâches à l'aide d'un algorithme classique d'ordonnancement mono-processeur.

Algorithme	Répartition équitable des ressources	Respect des dates d'échéance
First-In First-Out (FIFO)	non	non
Round-Robin (RR)	oui	non
Completely Fair Scheduler (CFS)	oui	non
Rate Monotonic Scheduling (RMS)	non	oui
Earliest Deadline First (EDF)	non	oui

TABLE 4.2 – Quelques algorithmes d'ordonnancement de tâches généralement utilisés.

4.4.1 Algorithmes généralement utilisés

Cette section ne traite que du problème d'ordonnancement de tâches : l'usage du terme priorité doit être compris ici comme faisant référence à la priorité d'exécution d'une tâche sur le cœur (fixée par l'ordonnanceur) et non à la priorité d'accès d'un cœur au bus (fixée par l'arbitre de bus) comme dans le reste de ce document.

Le tableau 4.2 liste quelques algorithmes couramment utilisés dans le domaine de l'ordonnancement de tâches. Nous ne traitons pas ici de *First-In First-Out*, *Round-Robin*, ou de *Completely Fair Scheduler*. En effet, ces différents algorithmes d'ordonnancement ne permettent pas d'assurer le respect des contraintes temporelles dans un système temps-réel strict. Cependant, il existe également des algorithmes permettant de prendre en compte la périodicité ou les dates d'échéance des tâches lors de leur ordonnancement, tels que *Rate Monotonic scheduling* ou *Earliest Deadline First*.

L'algorithme *Rate Monotonic scheduling* (RMS) a été présenté par Liu et Layland en 1973. Il permet de vérifier l'ordonnançabilité d'un ensemble de tâches périodiques à l'aide d'un simple calcul [63]. RMS attribue des priorités d'exécution fixes aux différentes tâches en fonction de leurs périodes : la tâche ayant la plus faible période bénéficie du niveau de priorité le plus élevé.

L'algorithme *Earliest Deadline First* (EDF) est quant à lui plus récent [19]. Dans un ordonnanceur utilisant EDF, les priorités d'exécution sont inversement proportionnelles aux dates d'échéances des tâches : lorsque plusieurs tâches sont arrivées en même temps, la tâche devant être terminée le plus tôt bénéficie de la plus haute priorité. Une comparaison détaillée des algorithmes *Rate Monotonic scheduling* et *Earliest Deadline First* a été dressée dans [15]. Un ensemble de tâches n'est garanti ordonnançable avec RMS que lorsque l'utilisation du processeur est inférieure à $ln(2) \approx 0,69$. Cette condition est suffisante mais non nécessaire : certains ensembles de tâches présentant un taux d'utilisation supérieur peuvent être ordonnançables, mais pas tous. EDF peut quant à lui offrir cette garantie pour des charges de travail avec des taux d'utilisation allant jusqu'à 1. Les tests d'ordonnançabilité nécessaires sont présentés en section 4.4.2.

Une autre caractéristique qui rend EDF plus intéressant que RMS dans le cas présent est qu'il peut générer un ordonnancement non-préemptif des tâches (NP-EDF), ce qui réduit significativement la difficulté d'estimation du temps d'exécution pire-cas des tâches. En effet, la préemption du processeur par l'ordonnanceur afin de laisser la main à une autre tâche introduit des délais supplémentaires dans l'exécution des tâches. Ceux-ci sont notamment dus à la commutation de contexte du processeur : sauvegarde puis chargement des registres et des caches, vidage puis ré-amorçage du pipeline.

Cependant, l'analyse de temps d'exécution pire-cas et l'ordonnancement sont deux étapes non simultanées et indépendantes l'une de l'autre. Durant l'analyse statique, il est impossible de connaître le nombre et les dates d'occurrence de ces interférences. Par conséquent, nous considérons que l'exécution de chaque tâche se fait d'un seul bloc, c'est-à-dire qu'il n'y a pas de préemption possible entre tâches. Il est à noter que la possibilité d'un ordonnancement non-préemptif est généralement appréciée dans les systèmes embarqués temps-réel, certains traitements ne tolérant pas d'interruption par un évènement extérieur.

4.4.2 Ordonnancement avec *Earliest Deadline First* non-préemptif

La robustesse des tests d'ordonnancement repose sur la connaissance préalable des durées des tâches à exécuter. Dans le contexte présent, nous pouvons garantir que le temps d'exécution *réel* de chaque tâche t_i sera inférieur ou égal à C_i puisque cette valeur est un majorant issu de l'analyse statique de temps d'exécution pire-cas effectuée au préalable. Si l'ordonnancement des tâches généré avec ces valeurs robustes est valide, alors il le sera quels que soient les *scenarii* d'exécution des tâches (différents jeux d'entrée, conflits d'accès aux ressources partagées, etc.).

La quantité de calculs nécessaires à la mise en œuvre d'un ordonnancement avec NP-EDF rend assez compliquée l'implémentation matérielle de cet algorithme. Les tests sont donc réalisés *hors-ligne*. Ceci n'est pas gênant dans le cas présent puisque l'ensemble des tâches à exécuter sur le processeur, leurs différents temps d'exécution et l'allocation des tâches aux cœurs sont déjà déterminés *a priori*. Il est à noter que n'importe quel autre algorithme d'ordonnancement non-préemptif mono-processeur pourrait convenir, cependant NP-EDF est largement utilisé et est reconnu comme offrant de bonnes performances.

Tout sous-ensemble de tâches τ_k est considéré comme ordonnnançable avec EDF non-préemptif si les deux conditions suivantes sont satisfaites [47] :

$$\sum_{t_i \in \tau_k} \frac{C_i}{P_i} \leq 1 \tag{4.26}$$

$$\forall t_i \in \tau_k \mid \forall L : P_1 < L \leq P_i,\ L \geq C_i + \sum_{j=1}^{i-1} \lfloor \frac{L-1}{P_j} \rfloor . C_j \tag{4.27}$$

Dans les résultats présentés en section 4.5, la vérification des conditions d'ordonnançabilité avec NP-EDF a été dans un premier temps effectuée à l'aide de l'outil CHEDDAR [1] développé par le laboratoire LISyC (Université de Bretagne Occidentale). Dans un second temps nous avons implémenté directement les formules des équations 4.26 et 4.27 dans notre algorithme afin d'améliorer les délais d'obtention des solutions valides.

4.5 Résultats expérimentaux

Nous avons cherché à mesurer l'impact des mécanismes d'arbitrage à deux niveaux *Group Round Robin* et *Geometric Group Latencies* sur le taux d'utilisation global d'un

1. `http://beru.univ-brest.fr/~singhoff/cheddar/`

processeur multi-cœurs exécutant une charge de travail hétérogène complexe, où le nombre de tâches dépasse le nombre de cœurs disponibles. Les tâches sont allouées aux cœurs suivant l'algorithme présenté en section 4.3.2. Nous présentons ici les résultats obtenus.

4.5.1 Méthodologie

Nous avons utilisé une charge de travail de test constituée de 32 tâches. Il s'agit en réalité d'un ensemble de 16 tâches différentes (listées en Annexe dans le tableau A.1) qui sont dupliquées une fois : nous avons remarqué que la distribution des tâches sur les différents cœurs est plus homogène avec des tâches deux fois plus nombreuses mais ayant un taux d'utilisation deux fois moindre, ce qui revient au final à la même charge totale. Les modalités de calcul des temps d'exécution des tâches sont les mêmes qu'au chapitre précédent (cf. tableau 3.2 et section 3.4.2 pour les caractéristiques des cœurs). Notamment, nous utilisons toujours un processeur à 8 cœurs et le même type de bus, afin de garder les mêmes latences. Notre algorithme évalue toutes les possibilités d'allocation pour toutes les configurations de protocole possibles avec un nombre de cœurs disponibles compris entre 1 et 8. Les temps d'exécution de chaque tâche ont donc été calculés pour 17 valeurs de latences différentes, comprises entre 10 et 217 cycles. Nous avons toujours considéré un cache de données parfait (*Data Always-Hit*).

La détermination des dates d'échéance des tâches, qui sont fixes, se fait à l'aide du taux d'utilisation unitaire u correspondant à une configuration de référence où huit cœurs accèdent à un bus arbitré par le protocole *Round-Robin*. Soit L la valeur de la latence d'accès au bus de référence observée avec huit cœurs et un bus arbitré par RR. Les valeurs de C_i^L correspondent au temps d'exécution pire-cas de chaque tâche t_i avec une latence L et sont fournies par la méthode décrite au chapitre précédent. La période P_i (égale à la date d'échéance implicite D_i) de chaque tâche est calculée par la formule suivante :

$$\forall i \in [0, n-1],\ P_i = \frac{C_i^L}{u} \tag{4.28}$$

A la différence du chapitre précédent, où nous avons pu dans un second temps retrouver les meilleures allocations de tâches aux cœurs sans avoir à évaluer toutes les combinaisons possibles, il n'est pas possible ici de déterminer à l'avance une stratégie optimale en fonction des caractéristiques des tâches. Ceci est dû au fait que l'optimisation de la solution du système d'équations linéaires ne peut se faire que suivant un seul critère, le taux d'utilisation des cœurs. Néanmoins, notre algorithme ne parcourt pas tout l'espace des solutions : les combinaisons de tâches proposées par le solveur ILP vont du plus faible coefficient de charge processeur vers le plus important, ce qui garantit bien sûr qu'il n'existe pas de solution *valide* (ordonnançable) ayant une charge globale plus faible, mais aussi que toutes les solutions présentant des taux d'utilisation plus élevés ne sont pas évaluées.

Afin d'apporter une réponse partielle à ce problème, nous proposons un mécanisme de *verrouillage* des allocations valides qui permet de restreindre très fortement le nombre d'itérations réalisées par l'algorithme avant d'atteindre une solution ordonnançable, au prix d'une augmentation relativement limitée de la charge globale.

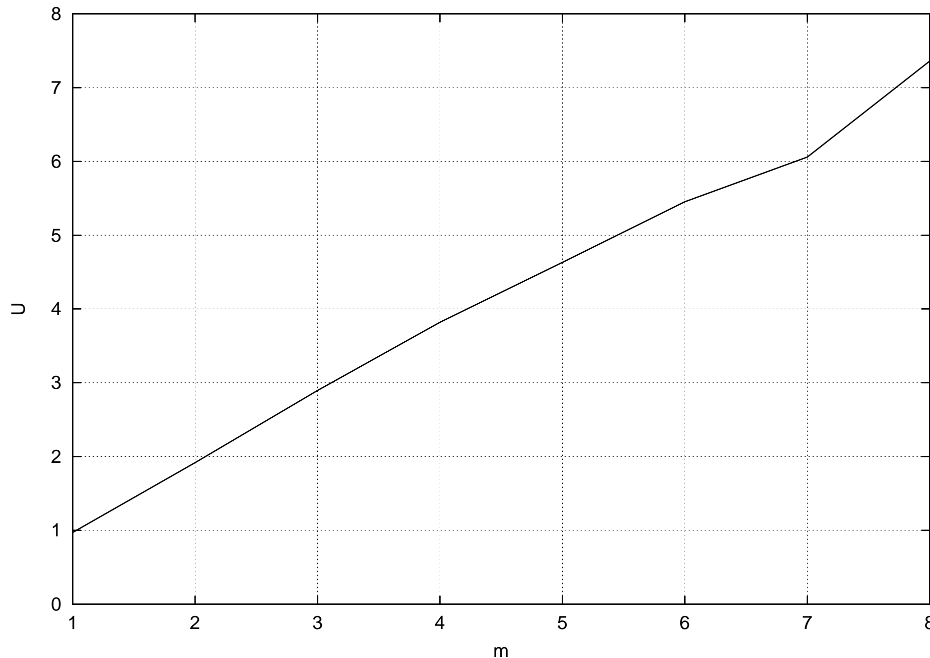

FIGURE 4.1 – Taux d'utilisation global U avec l'algorithme de *bin-packing*.

4.5.2 Evaluation de l'algorithme de *bin-packing*

A titre de référence et à fin de comparaison, nous avons tout d'abord évalué les performances de l'algorithme de *bin-packing* sur des charges de travail à taux d'utilisation statique, avec un arbitre de bus utilisant *Round-Robin*. Le taux d'utilisation unitaire u retenu pour l'exemple est égal à 0,23. Nous utilisons deux critères de tri des tâches : le taux d'utilisation υ_i et le temps d'exécution pire-cas C_i. Les tâches sont allouées avec l'heuristique *First-Fit Decreasing* à la fois lorsque le critère de tri retenu est celui du taux d'utilisation (la charge totale du cœur doit être inférieure ou égale à 1) mais aussi lorsque le tri se fait suivant les temps d'exécution pire-cas décroissants (la tâche est rangée sur le cœur le moins chargé en cycles processeur).

Les tests d'ordonnancement avec NP-EDF ont été intégrés à l'algorithme de *bin-packing*. A chaque essai de *rangement* d'une tâche sur un cœur, si la contrainte de l'heuristique est respectée, des tests sont lancés pour vérifier que l'ensemble de tâches alloué au cœur reste bien ordonnançable avec cette nouvelle tâche. Si ce n'est pas le cas, un essai d'allocation est lancé sur le cœur suivant, etc. Si la tâche ne peut être allouée à aucun des cœurs sans compromettre le respect des dates d'échéance des tâches qui s'exécutent déjà dessus, alors l'ensemble de tâches n'est pas ordonnançable.

Les figures 4.1 et 4.2 montrent le taux d'utilisation global du processeur U et le taux d'utilisation maximal par cœur $max(U_k)$ observés en fonction du nombre de cœurs utilisés m. Le seul critère de tri pour lequel nous avons reproduit les résultats ici est celui du taux d'utilisation u_i décroissant. En effet, lorsqu'on tente de ranger les tâches par temps d'exécution C_i décroissants l'algorithme ne trouve pas de solution, la charge totale U étant systématiquement supérieure au nombre de cœurs m.

En revanche, utiliser le taux d'utilisation comme critère d'allocation des tâches semble

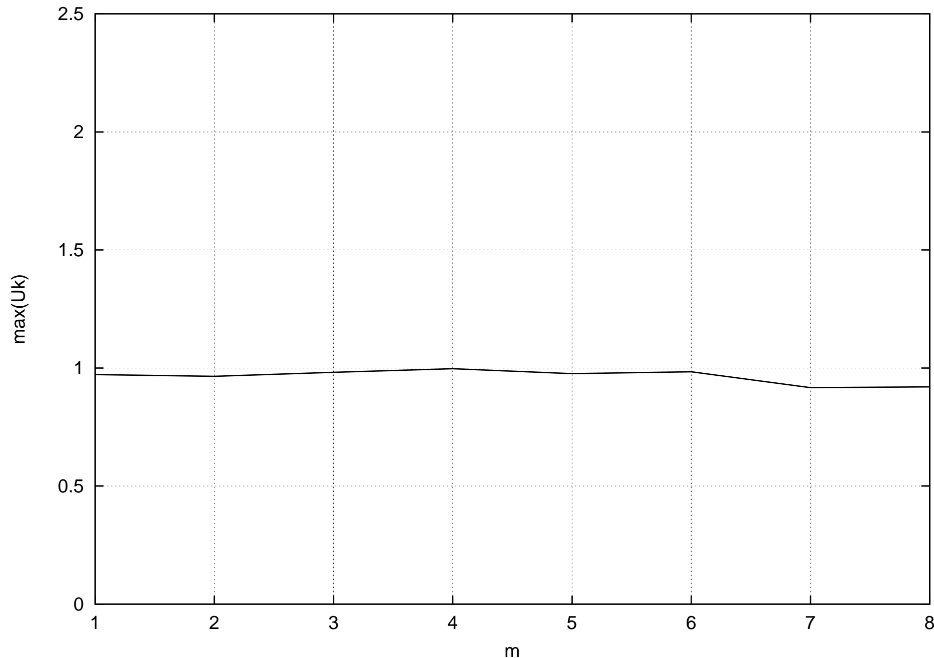

FIGURE 4.2 – Taux d'utilisation maximal par cœur $max(U_k)$ avec l'algorithme de *bin-packing*.

permettre d'équilibrer au mieux les charges des différents cœurs, même si celles-ci restent très proches de la limite supérieure : le caractère statique de l'arbitrage de bus et les latences importantes qu'il induit annulent quasiment le gain apporté par l'apport de cœurs supplémentaires.

Par ailleurs, malgré l'intégration des tests d'ordonnancement dans l'algorithme de *bin-packing*, lorsque u est égal à 0,23 seule la configuration avec $m = 8$ cœurs et utilisant l'allocation par v_i a pu aboutir à une solution ordonnançable avec NP-EDF. Les autres solutions montrées dans les deux figures sont des solutions par défaut où une ou plusieurs tâches ne respectent pas leurs échéances temporelles.

La suite de ce chapitre s'intéresse aux résultats obtenus avec notre nouvel algorithme d'allocation de tâches à taux d'utilisation variable.

4.5.3 Recherche d'une solution valide

La figure 4.3 présente l'évolution de différents paramètres en fonction du nombre d'itérations réalisées par l'algorithme 4.1 sur un exemple comportant un total de six cœurs disponibles et utilisant la configuration de protocole GGL-{1,2,3}. Le taux d'utilisation global U est représenté sur le graphe par une courbe pleine, la courbe en croix-pointillés suit la quantité de tâches non ordonnançables, et la courbe en pointillés montre le nombre d'ensembles de tâches non encore valides.

L'algorithme utilise ici le verrouillage des ensembles de tâches valides. Le taux d'utilisation global U augmente au fur à mesure des solutions fournies par le solveur : à la première itération il est égal à 4,66, alors qu'à la trente-sixième itération (celle aboutissant à une solution valide) il est passé à 4,79.

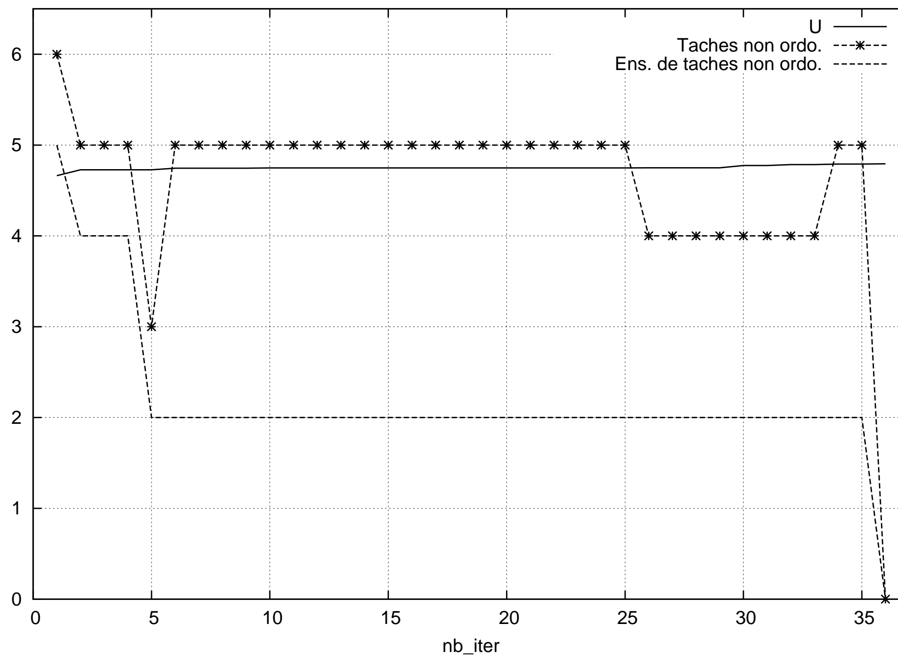

FIGURE 4.3 – Évolution du taux d'utilisation global U, du nombre de tâches non ordonnançables et du nombre d'ensembles de tâches non ordonnançables en fonction du nombre d'itérations de l'algorithme 4.1 et en considérant la configuration GGL-{1,2,3}.

On peut observer qu'à la première itération de l'algorithme il y a six tâches non ordonnançables, puis ce nombre se stabilise autour de cinq tandis que seuls deux ensembles de tâches restent non valides. Par ajouts successifs de contraintes, l'algorithme parvient en une trentaine d'itérations à résoudre les conflits existant entre les tâches restant à ordonnancer. Le graphe 4.3 illustre bien le fonctionnement du principe de *whitelist* : le nombre d'ensembles de tâches non ordonnançables ne peut aller qu'en diminuant, puisque tous ceux qui sont valides sont automatiquement verrouillés. Ceci n'est en revanche pas valable pour le nombre de tâches : le fait de basculer une tâche d'un ensemble à un autre peut créer des conflits supplémentaires, ce qui n'est pas prévisible par le solveur puisque les tests d'ordonnancement se font de manière externe.

4.5.4 Charges globales minimales atteintes

Les résultats présentés dans cette partie ont été obtenus en considérant un taux d'utilisation unitaire u égal à 0,21. En effet, les tests ont montré que notre algorithme ne permet pas de trouver une allocation des tâches sur huit cœurs valide en utilisant un arbitrage de bus suivant *Round-Robin* lorsque u est supérieur à 0,21. Conserver cette valeur permet donc d'établir un point de comparaison entre les différentes politiques de bus et d'évaluer l'amélioration de performances apportée par GRR et GGL.

Configuration			GRR		GGL	
N_1	N_2	N_3	$min(U)$	nb_iter	$min(U)$	nb_iter
6	-	-	*N/O*	-	-	-
1	5	-	**4,89**	56	-	-
2	4	-	*N/O*	-	-	-
1	1	4	**4,89**	1093	**4,65**	72
1	2	3	*N/O*	-	4,79	54
2	1	3	*N/O*	-	*N/O*	-
3	1	2	*N/O*	-	5,48	42
4	1	1	**4,89**	1093	5,23	**28**
7	-	-	*N/O*	-	-	-
1	6	-	5,02	55	-	-
2	5	-	*N/O*	-	-	-
3	4	-	*N/O*	-	-	-
1	1	5	4,93	276	4,70	63
1	2	4	*N/O*	-	4,82	50
1	3	3	5,44	121	5,02	55
2	1	4	*N/O*	-	*N/O*	-
2	2	3	*N/O*	-	*N/O*	-
3	1	3	5,44	121	5,51	397
4	1	2	*N/O*	-	5,80	200
8	-	-	***6,72***	380	-	-
1	7	-	5,16	**51**	-	-
2	6	-	5,51	**7290**	-	-
3	5	-	*N/O*	-	-	-
1	1	6	4,97	1382	4,76	62
1	2	5	5,41	809	4,86	50
1	3	4	5,46	240	5,04	56
2	1	5	5,41	809	5,32	1039
2	2	4	*N/O*	-	5,40	**4224**
3	1	4	5,46	240	5,53	38
3	2	3	6,20	115	*N/O*	-
4	1	3	5,46	240	**5,82**	1217
5	1	2	5,41	809	*N/O*	-
		Moyenne	*5,37*	*824,33*	*5,17*	*477,94*

TABLE 4.3 – Taux d'utilisation global minimum $min(U)$ obtenu pour chaque configuration avec l'algorithme 4.1 (sans prise en compte de la *whitelist*) et un taux d'utilisation par tâche u égal à 0,21.

Sans verrouillage des allocations valides

Le tableau 4.3 présente les résultats d'allocation obtenus en utilisant seulement l'interdiction des ensembles de tâches non ordonnançables (*blacklist*). Etant donné que les paramètres énoncés ci-dessus n'ont pas rendu possible de trouver une allocation valide des tâches sur cinq cœurs ou moins, seules les configurations utilisant au minimum six cœurs sont listées ici. Les trois premières colonnes correspondent au nombre de cœurs N_i dans chacun des groupes de priorité G_i. La quatrième (respectivement sixième) colonne donne pour chaque configuration du protocole GRR (resp. GGL) le taux d'utilisation global minimum $min(U)$ pour lequel l'algorithme a déterminé une solution valide. Les cinquième et septième colonnes donnent le nombre d'itérations de l'algorithme ayant été nécessaires à l'obtention de la solution.

Comme dans les résultats présentés au chapitre précédent, les protocoles GRR et GGL donnent les mêmes latences de bus pour les cas où il n'y a qu'un ou deux groupes de priorité. De même, pour un protocole donné certaines configurations sont équivalentes, par exemple GRR-{1,3,3} et GRR-{3,1,3} aboutissent au même schéma d'arbitrage et aux mêmes résultats. Lorsque nous décomptons le nombre de configurations pour lesquelles l'algorithme a trouvé une solution valide, nous ne prenons pas en compte ces doublons. Les cases indiquant N/O correspondent aux cas où l'algorithme n'a pas trouvé de solution ordonnançable.

La valeur de référence (en italique) correspond à un arbitre de bus suivant *Round-Robin* sur huit cœurs (représenté dans le tableau par la ligne GRR-{8}). Dans cette configuration, l'algorithme d'allocation des tâches aux cœurs a trouvé une solution valide ayant une charge globale $min(U)$ égale à 6,72. En revanche, en utilisant le protocole GRR, la charge globale du système peut descendre à 4,89 sur seulement six cœurs avec les configurations GRR-{1,5} et GRR-{1,1,4} (ou GRR-{4,1,1}), ce qui représente une diminution de 27,2% par rapport à RR. Lorsque l'algorithme évalue les différentes possibilités du protocole GGL, la charge du système peut être réduite à 4,65 sur six cœurs avec la configuration GGL-{1,1,4}, ce qui équivaut cette fois-ci à une amélioration de 30,8% par rapport à la valeur de référence. De manière générale, on peut remarquer que les charges obtenues avec GRR et GGL sont systématiquement inférieures à la valeur de référence, car pour chaque configuration la valeur donnée est celle de la solution ordonnançable ayant la charge la plus faible (parmi tout l'espace des solutions valides).

Suivant les configurations, la performance de l'algorithme peut varier grandement. Avec *Group Round Robin*, de 51 à 7290 itérations ont été en moyenne nécessaires pour déterminer une solution ordonnançable, alors que le système de contraintes linéaires a dû être enrichi de 28 à 4224 fois dans le cas de *Geometric Group Latencies*. Si l'on additionne tous les calculs nécessaires au parcours de toutes les configurations d'arbitre possibles, ce sont pas moins de 19.515 itérations qui ont été réalisées par l'algorithme.

Au final, les solutions valides aux configurations de GRR présentent une moyenne des charges globales minimales obtenues de 5,37, tandis que dans le cas de GGL cette valeur est égale à 5,17. En moyenne, le nombre d'itérations nécessaires à l'obtention une solution valide est d'environ 824 pour le protocole GRR et de 477 pour GGL. Si on comptabilise pour les deux protocoles les configurations à un et deux groupes, il y a dans le cas de GRR 12 configurations ayant abouti à une solution valide tandis qu'il y en a 20 pour le protocole GGL.

Finalement, on peut noter que GGL surpasse légèrement GRR tant en diminution de la charge du processeur et en performances de l'algorithme qu'en quantité de configurations ordonnançables.

Avec verrouillage des allocations valides

Le tableau 4.4 présente les résultats d'allocation obtenus en utilisant à la fois l'interdiction des ensembles de tâches non ordonnançables et le verrouillage des ensembles de tâches valides (*blacklist + whitelist*). La présentation des résultats se fait sous la même forme que pour le tableau 4.3. Ici, la valeur de référence pour le taux d'utilisation global minimal du processeur $min(U)$ avec un arbitrage de bus suivant *Round Robin* est toujours de 6,72. Ceci est normal puisque dans le cas d'un ensemble de tâches ayant un taux d'utilisation statique, la charge totale du système reste constante quels que soient les cœurs exécutant les combinaisons de tâches.

En utilisant le protocole *Group Round Robin*, l'algorithme trouve cette fois-ci une solution valide permettant d'atteindre une charge globale de 5,01 avec sept cœurs et la configuration GRR-{1,1,5}, ce qui représente une amélioration de 25,4% par rapport à la valeur de référence. Lorsque l'algorithme évalue les différentes possibilités du protocole GGL, la charge du système peut être réduite à 4,70, toujours sur sept cœurs et avec la configuration GGL-{1,1,5}, ce qui équivaut à une réduction de 30,1% du taux d'utilisation constaté avec RR.

Suivant les configurations, le nombre d'itérations effectuées par l'algorithme est plus ou moins élevé. Avec *Group Round Robin*, de 5 à 101 itérations ont été nécessaires pour déterminer une solution valide, alors qu'avec *Geometric Group Latencies* de 6 à 111 itérations ont dû être effectuées.

Au final, les solutions valides aux configurations de GRR présentent une moyenne des charges globales minimales de 5,54, tandis que dans le cas du protocole GGL cette valeur est égale à 5,37. En moyenne également, le nombre d'itérations effectuées avant l'obtention d'une solution valide est d'environ 27 pour le protocole GRR et de 34 pour GGL. Si on comptabilise pour les deux protocoles les configurations à un et deux groupes, il y a dans le cas de GRR 16 configurations ayant abouti à une solution valide tandis qu'il y en a 21 pour le protocole GGL. En additionnant tous les calculs nécessaires au parcours de toutes les configurations d'arbitre différentes, ce sont 917 itérations qui ont été réalisées par l'algorithme dans le cas présent.

Dans le cadre d'un algorithme utilisant à la fois l'interdiction des combinaisons de tâches non ordonnançables et le verrouillage des allocations valides, on remarque que si GGL surpasse légèrement GRR lorsqu'il s'agit de diminuer la charge du processeur ou d'obtenir plus de configurations ordonnançables, c'est par contre GRR qui donne les meilleures performances en termes de rapidité d'obtention d'une solution.

Si l'on compare les résultats obtenus sans verrouillage des allocations valides à ceux montrés ici, on remarque qu'ils sont générés beaucoup plus rapidement dans le second cas. Le nombre moyen d'itérations nécessaires diminue de 96,7% pour GRR et de 92,8% pour GGL. Le prix à payer de cette amélioration importante de performances est que les solutions atteintes sont moins optimales : par rapport aux premiers résultats, la charge moyenne est augmentée de 3,1% avec GRR et de 3,9% avec GGL. Ceci étant, la plus grande rapidité de l'algorithme avec verrouillage permet de générer des quantités beaucoup plus

Configuration			**GRR**		**GGL**	
N_1	N_2	N_3	$min(U)$	nb_iter	$min(U)$	nb_iter
6	-	-	*N/O*	-	-	-
1	5	-	5,07	7	-	-
2	4	-	*N/O*	-	-	-
1	1	4	*N/O*	-	*N/O*	-
1	2	3	*N/O*	-	4,79	36
2	1	3	*N/O*	-	*N/O*	-
3	1	2	*N/O*	-	*N/O*	-
4	1	1	*N/O*	-	*N/O*	-
7	-	-	*N/O*	-	-	-
1	6	-	5,02	6	-	-
2	5	-	5,61	10	-	-
3	4	-	*N/O*	-	-	-
1	1	5	**5,01**	29	**4,70**	62
1	2	4	5,37	12	4,82	34
1	3	3	5,44	8	5,02	**6**
2	1	4	5,37	12	*N/O*	-
2	2	3	5,98	82	*N/O*	-
3	1	3	5,44	8	5,65	16
4	1	2	5,37	12	5,90	43
8	-	-	***6,72***	**101**	-	-
1	7	-	5,30	28	-	-
2	6	-	5,86	9	-	-
3	5	-	6,15	13	-	-
1	1	6	5,03	18	4,76	62
1	2	5	5,30	75	4,86	7
1	3	4	5,46	**5**	5,05	32
2	1	5	5,30	75	*N/O*	-
2	2	4	6,15	10	5,61	**111**
3	1	4	5,46	**5**	5,72	20
3	2	3	6,35	19	**6,42**	18
4	1	3	5,46	**5**	5,95	28
5	1	2	5,30	75	5,86	10
	Moyenne		*5,54*	*27,13*	*5,37*	*34,64*

TABLE 4.4 – Taux d'utilisation global minimum $min(U)$ obtenu pour chaque configuration avec l'algorithme 4.1 (avec prise en compte de la *whitelist*) et un taux d'utilisation par tâche u égal à 0,21.

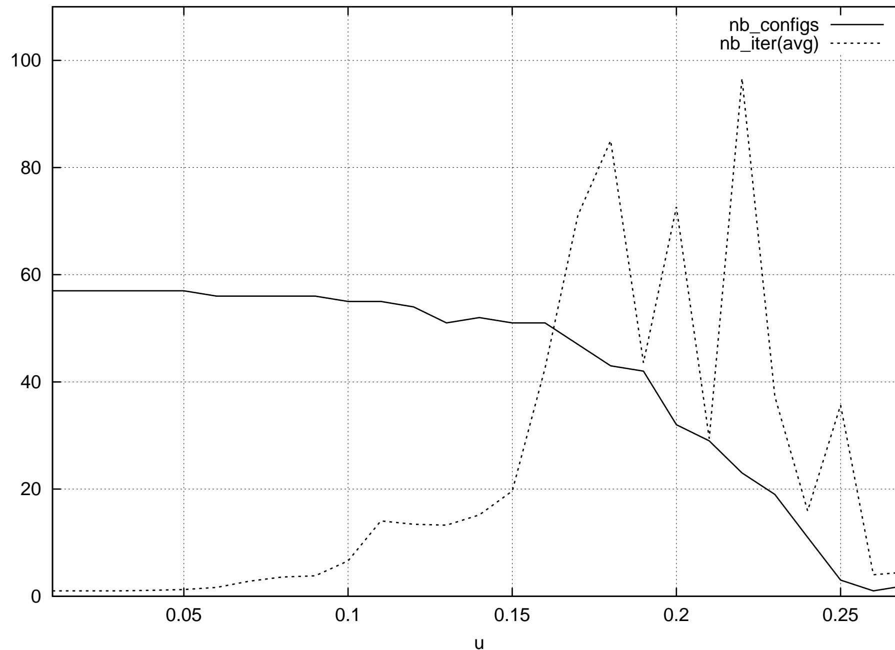

FIGURE 4.4 – Nombre de configurations ayant au moins une solution ordonnançable et nombre moyen d'itérations de l'algorithme en fonction du taux d'utilisation par tâche u.

importantes de résultats (cf. section 4.5.5).

Dans les deux tableaux, il semble raisonnable d'affirmer que les meilleurs résultats obtenus avec GGL sont dûs au plus grand nombre de configurations différentes qu'il est possible de générer à partir d'un même nombre de cœurs, ce qui favorise une adaptation fine de la latence de bus aux besoins de chaque tâche. Quant à la variabilité des performances de l'algorithme d'allocation, celle-ci semble s'expliquer par la nature de l'ensemble des tâches exécutées et par les conflits d'ordonnancement qui peuvent apparaître entre elles, plutôt que par la configuration d'arbitre retenue.

4.5.5 Ordonnançabilité en fonction du taux d'utilisation unitaire

Comme il n'est pas possible de reproduire ici l'ensemble des résultats obtenus avec notre algorithme de résolution en testant plusieurs valeurs de taux d'utilisation *unitaire* différentes, nous avons plus simplement cherché à évaluer la corrélation entre cette variable et les possibilités d'obtention d'une solution valide.

Sur la figure 4.4, la courbe pleine présente l'évolution du nombre de configurations $nb_configs$ ayant au moins une allocation de tâches aux cœurs ordonnançable en fonction du taux d'utilisation par tâche u. Les valeurs de u testées sont comprises dans l'intervalle $[0,01 : 0,3]$ par pas de 0,01, des résultats ayant été trouvés pour $u \leq 0,27$.

Il y a 57 configurations différentes de cœurs dans les groupes possibles au total. Tant que la valeur de u demeure inférieure à 0,17, notre algorithme est capable de déterminer des solutions valides pour au moins 50 configurations différentes. Le nombre de solutions décroît ensuite rapidement, notamment parce qu'il n'est plus possible d'ordonnancer la charge de travail sur un nombre réduit de cœurs. Lorsque u vaut 0,27 il reste encore deux

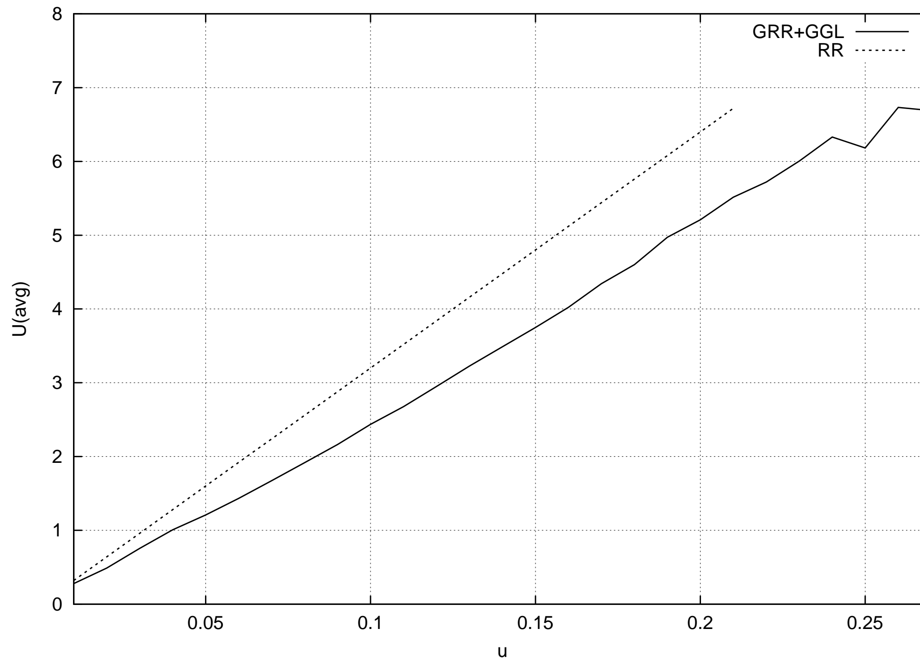

FIGURE 4.5 – Taux d'utilisation global moyen observé U_{avg} et charge de référence en fonction du taux d'utilisation par tâche u.

configurations de bus pouvant aboutir à une allocation de tâches valide.

Sur cette même figure, la courbe en pointillés présente quant à elle l'évolution du nombre moyen d'itérations nb_iter_{avg} nécessaires à l'obtention d'une solution valide, en fonction du taux d'utilisation par tâche u. La valeur nb_iter_{avg} est calculée en faisant la moyenne des nombres d'itérations de l'algorithme nb_iter obtenus pour les configurations ayant une solution valide.

On peut observer qu'en moyenne le nombre d'itérations est peu élevé pour des valeurs de u faibles, ce qui laisse à penser que dans ces conditions il y a peu de conflits de dates d'échéance entre les tâches et qu'ainsi l'algorithme arrive facilement à une solution. Ceci est d'ailleurs corroboré sur le même graphe par le nombre élevé de configurations ayant obtenu une solution valide. Les performances de l'algorithme se dégradent ensuite en même temps que le nombre de configurations ayant abouti à une solution diminue.

La figure 4.5 présente l'évolution du taux d'utilisation global moyen U_{avg} en fonction du taux d'utilisation par tâche u. La courbe en pointillés sert de référence et montre les taux d'utilisation atteints avec *Round-Robin* et l'algorithme de *bin-packing*. Il est à noter que les taux atteints en utilisant RR avec notre algorithme sont strictement identiques (puisqu'ils ne dépendent pas du placement des tâches sur les cœurs). La courbe pleine désigne quant à elle les valeurs de U_{avg}, calculées pour chaque valeur de u en faisant la moyenne des taux d'utilisation globaux minimum $min(U)$ obtenus pour les configurations ayant une solution valide.

On remarque que la charge moyenne suit elle aussi une évolution quasi-linéaire en fonction du taux d'utilisation unitaire. Plus la valeur de u augmente, plus l'écart entre la charge de référence et les valeurs déterminées par notre algorithme de résolution de systèmes de contraintes linéaires est important. A titre d'exemple, avec un taux d'utilisation

par tâche de 0,19, un arbitrage de bus avec RR atteint un coefficient de charge de 6,08, tandis qu'il n'est que de 4,97 avec notre algorithme.

Finalement, même en utilisant les 8 cœurs disponibles il devient impossible d'utiliser un arbitrage de bus fonctionnant suivant *Round-Robin* lorsque u est supérieur à 0,21, alors que les arbitres GRR et GGL peuvent atteindre une valeur de u égale à 0,27. Ceci prouve que l'utilisation conjointe de notre algorithme et des protocoles GRR et GGL permet d'ordonnancer des charges de travail plus importantes en gardant une configuration matérielle identique (quantité et caractéristiques des cœurs).

Ce graphe montre également que des taux d'utilisation supérieurs à 7 semblent difficiles à atteindre en pratique sur un processeur octo-cœurs tout en maintenant les ensembles de tâches ordonnançables. Cette caractéristique dépend par ailleurs fortement de la nature de l'ensemble de tâches à ordonnancer : étant non préemptibles et générant des grandes durées d'attente pour les autres tâches, les tâches très longues ont tendance à faire chuter le taux d'utilisation des cœurs.

Conclusion

Nous avons étendu dans ce chapitre le champ d'application des arbitres à deux niveaux à des plateformes exécutant des charges de travail complexes, où plusieurs tâches ayant des contraintes temps-réel strict sont allouées à chaque cœur. La variabilité des temps d'exécution nécessite la mise au point d'un algorithme pouvant partitionner des ensembles de tâches à taux d'utilisation dynamique. La solution que nous avons proposée repose sur une représentation du problème d'allocation sous forme de programme linéaire. Notre algorithme évalue toutes les configurations possibles des arbitres *Group Round Robin* et *Geometric Group Latencies*. Il permet de déterminer un couple configuration / allocation de tâches qui minimise la charge globale du système, afin de pouvoir exécuter simultanément d'autres tâches n'ayant pas de contraintes temporelles strictes. Ceci se traduit aussi par une diminution du nombre de cœurs nécessaires, ce qui peut permettre par exemple de séparer complètement les cœurs réservés aux tâches temps-réel de ceux exécutant des tâches de criticité moindre.

De manière globale, l'utilisation conjointe de notre algorithme et des arbitres à deux niveaux permet d'améliorer significativement les performances de la plateforme. Les résultats montrent qu'en considérant un cache de données parfait, le protocole GGL permet ici de réduire de 30,8% la charge totale du système dans le meilleur des cas et ceci par rapport à une allocation des tâches par *bin-packing* avec un arbitrage de bus classique. Cette valeur est par ailleurs comparable aux résultats présentés au chapitre précédent, où la somme des temps d'exécution des tâches a pu être réduite de 27,9% dans les mêmes conditions. Afin de raccourcir les délais des calculs nécessaires à l'obtention d'une solution valide, nous évaluons une variante de notre algorithme où les sous-ensembles de tâches ordonnançables sont verrouillées sur un cœur, ce qui permet de restreindre l'espace des solutions à explorer. La charge minimale obtenue dans ce cas est réduite de 30,1% par rapport à la plateforme de référence, ce qui est pratiquement équivalent à la première variante de l'algorithme. Dans le même temps, le nombre d'itérations nécessaires au parcours de toutes les configurations et à l'obtention de la meilleure solution est réduit de 95,3%. Les temps de calcul, non mesurés dans notre protocole expérimental, sont raccourcis dans un

ordre de grandeur comparable.

Finalement, nous montrons que notre mécanisme permet également d'ordonnancer des tâches plus lourdes (ou un nombre de tâches plus élevé). En effet, dans le protocole expérimental proposé, utiliser un arbitrage de bus suivant *Round-Robin* sur une plateforme à huit cœurs ne permet pas de garantir l'ordonnançabilité d'un ensemble de 32 tâches dont le taux d'utilisation unitaire dépasse 0,21. En revanche, en utilisant les protocoles GRR et GGL notre algorithme permet de déterminer des allocations valides jusqu'à un taux de 0,27, ce qui représente une augmentation de 22,2% de la capacité utile du processeur.

Conclusion

La vérification des contraintes temporelles d'un système temps-réel strict dépend de la connaissance du temps d'exécution pire-cas des tâches constituant l'application embarquée. L'utilisation de processeurs multi-cœurs est l'un des moyens actuellement mis en œuvre afin d'améliorer le niveau de performances des systèmes embarqués. Cependant, la détermination du temps d'exécution pire-cas d'une tâche sur ce type d'architecture est rendue difficile par le partage de certaines ressources entre les cœurs. Les conflits d'accès au bus d'interconnexion peuvent notamment rendre imprévisibles les délais de chargement d'une donnée depuis ou vers la mémoire centrale.

Contributions

Nous avons tout d'abord proposé un mécanisme d'arbitrage de bus à deux niveaux permettant d'offrir des niveaux de priorité différenciés aux requêtes des différents cœurs. Les latences pire-cas d'accès au bus garanties par ce dispositif sont totalement prévisibles et ne dépendent pas du comportement des autres tâches, ce qui permet de les intégrer à l'analyse statique de temps d'exécution pire-cas. Les deux politiques de gestion de l'arbitre introduites, *Group Round Robin* et *Geometric Group Latencies*, définissent des groupes de priorité auxquels appartiennent un ou plusieurs cœurs qui partagent ainsi les mêmes délais d'accès au bus. Ce mécanisme permet de s'adapter à des ensembles de tâches hétérogènes : la priorité de chaque cœur est ajustée en fonction des besoins en bande passante de la tâche qu'il exécute.

Identifier le scénario menant à la plus grande amélioration des temps d'exécution pire-cas peut être rédhibitoire, en raison de l'étendue des possibilités à évaluer (attribution des tâches aux cœurs et répartition des cœurs dans les groupes). Nous avons donc introduit un critère d'allocation : la *sensibilité* du temps d'exécution pire-cas à la latence de bus. Cette valeur permet d'identifier les tâches qui bénéficient le plus de latences d'accès au bus réduites. En allouant les tâches présentant une grande sensibilité du temps d'exécution aux cœurs ayant le niveau de priorité le plus élevé, les performances optimales de notre mécanisme sont systématiquement atteintes.

Nous avons ensuite étendu le champ d'application de notre arbitre à deux niveaux à des charges de travail complexes, où le nombre de tâches ayant des contraintes temps-réel strictes dépasse le nombre de cœurs disponibles. Nous avons utilisé une stratégie de partitionnement de la charge en sous-ensemble de tâches exécutés chacun par un unique cœur et sur lequel un test d'ordonnancement mono-processeur peut être réalisé. L'utilisation des politiques GRR et GGL induit que les tâches ont des taux d'utilisation du processeur dynamiques, c'est-à-dire variant suivant le cœur sur lequel elles sont exécutées. Les algo-

rithmes d'allocation les plus répandus ne permettent pas de traiter ce cas de figure.

Nous avons proposé un algorithme effectuant une représentation du problème d'allocation des tâches aux cœurs sous forme de programme linéaire. Il permet d'identifier, pour chaque configuration possible de l'arbitre, une allocation de tâches aux cœurs qui minimise la charge totale du processeur et le nombre de cœurs utilisés. L'ordonnançabilité des ensembles de tâches ainsi générés est garantie. Dans un second temps, nous avons évalué une variante de notre algorithme permettant de réduire grandement les délais de calcul nécessaires à l'obtention d'une solution valide.

Perspectives

Nous avons listé quelques perspectives intéressantes de problèmes non résolus ou d'extension des méthodes présentées ici et pouvant faire l'objet de travaux futurs.

Optimisation multi-critères de l'allocation

La méthode d'allocation de tâches aux cœurs présentée en section 4.3.2 repose sur un algorithme itératif qui évalue l'ordonnançabilité de combinaisons de tâches en allant de la plus faible charge totale possible vers la plus élevée. Nous avons utilisé la charge totale comme fonction à minimiser dans le programme linéaire, afin d'être sûr d'obtenir les solutions menant au plus grand potentiel d'amélioration des performances. Ceci entraîne l'exploration complète de l'espace des solutions non ordonnançables ayant un facteur de charge totale inférieur à la solution valide. Le coût de cette exploration est non négligeable en termes de durées de calcul.

L'optimisation de la solution d'un système d'équations linéaires ne peut se faire que suivant un seul critère. Il serait pourtant intéressant de pouvoir introduire simultanément un ou plusieurs critères d'optimisation supplémentaires, comme ceux ayant trait aux caractéristiques des tâches, et notamment la *sensibilité* du temps d'exécution pire-cas à la latence de bus. Nous avons vu qu'allouer les tâches ayant les valeurs de sensibilité les plus fortes aux cœurs de priorité supérieure permet d'aboutir à la combinaison optimale lorsque le nombre de tâches à exécuter est inférieur ou égal au nombre de cœurs. La seconde fonction linéaire à minimiser ici pourrait donc être égale à la somme des sensibilités des tâches pondérées de la latence du cœur sur lequel elles s'exécutent.

L'utilisation d'algorithmes de recherche multi-critères permet de se rapprocher plus rapidement d'*une* solution optimale. En effet, il n'existe généralement pas de solution pouvant satisfaire tous les critères à la fois. Les algorithmes évolutionnistes permettent cependant de trouver des solutions acceptables dans un contexte d'optimisation multi-critère [95]. Ils pourraient être utilisés en lieu et place du problème linéaire en nombres entiers.

Prise en compte de l'hétérogénéité des cœurs

La variabilité des temps d'exécution sur un processeur multi-cœurs n'est pas uniquement due à la différence entre les niveaux de service garantis aux cœurs lorsqu'ils accèdent aux ressources partagées. La différenciation des cœurs eux-mêmes est amenée à se répandre

de plus en plus dans les *systèmes sur puce* embarqués, dans un premier temps ceux exécutant des applications temps-réel souple (comme les tablettes tactiles ou les téléphones de nouvelle génération).

En 2011, ARM a proposé le processeur *Big.LITTLE* qui couple des cœurs *Cortex A7* et *Cortex A15* sur une même puce. La société nVidia a de son côté présenté l'architecture *Kal-El*, basée sur une technologie nommée *Variable Symmetric Multiprocessing* (vSMP) qui associe quatre cœurs principaux à un cœur *compagnon*. Dans les deux cas, un des cœurs présente une fréquence d'horloge moindre ou une micro-architecture plus simple et est destiné aux tâches de fond (par exemple, synchronisation du courrier électronique) [98]. Pendant que ce cœur lent est utilisé, les autres sont désactivés afin d'économiser la batterie de l'appareil. Ces architectures présentent l'avantage d'adapter continuellement la quantité de ressources matérielles disponibles aux besoins de l'ensemble de tâches exécuté.

L'approche d'ARM comprend également un modèle de *multiprocessing* hétérogène où tous les cœurs sont utilisés en même temps et où chaque tâche est allouée à l'un ou l'autre des types de cœurs en fonction de ses besoins ou du niveau de priorité qui lui est associé. Ceci rejoint le champ d'application de notre algorithme permettant de déterminer une allocation optimale des tâches sur des processeurs où le niveau de performance est différencié suivant les cœurs.

Cependant, comme dans n'importe quel processeur, le coût du transfert d'une tâche entre deux cœurs n'est jamais nul. En effet, il faut sauvegarder puis re-charger le contenu des registres et des caches, vider le pipeline du premier cœur puis ré-amorcer celui du second, et ainsi de suite. Par exemple, ARM annonce un délai de migration de 20.000 cycles qui doit être pris en compte pour déterminer s'il est ou non plus avantageux de transférer la tâche en cours d'exécution sur un cœur principal vers le cœur compagnon. De plus, la prise en compte des préemptions et des migrations dans le calcul de temps d'exécution pire-cas nécessite l'analyse conjointe de l'ensemble de tâches exécuté, comme dans [23].

Analyse d'applications parallèles

Les processeurs multi-cœurs permettent d'améliorer les performances des systèmes embarqués en exécutant simultanément des tâches aux caractéristiques différentes (temps d'exécution, criticité, etc.). Cependant, nous avons vu qu'il est difficile de répartir la charge de travail totale de manière homogène sur les différents cœurs. C'est pourquoi il apparaît nécessaire d'exploiter le *parallélisme de tâche* : chaque tâche est séparée en plusieurs processus légers s'exécutant en parallèle et se synchronisant pour échanger des données. Cependant, à l'heure actuelle les applications temps-réel peuvent encore difficilement tirer parti de cette technique. En effet, elles sont conçues pour être exécutées de manière séquentielle, et bien souvent les compilateurs effectuant une parallélisation automatique du code ne prennent pas en compte les contraintes temporelles des systèmes embarqués.

Le projet *parMERASA* [2] se propose de répondre au défi de la parallélisation d'applications temps-réel sur un processeur multi-cœur. Une description précise de la tâche parallélisée, par exemple sous forme de graphe de dépendances entre processus, est nécessaire

2. http://www.parmerasa.eu/

afin d'effectuer l'analyse statique de temps d'exécution pire-cas de l'ensemble de tâches exécuté. Une solution peut être de séparer les périodes de calcul des périodes de communication entre processus, afin d'identifier clairement les endroits où les tâches doivent se synchroniser. Dans le cas de processus parallèles dont les temps d'exécution ne sont pas identiques, il peut être intéressant de diminuer les latences d'accès aux ressources partagées subies par les processus critiques, c'est-à-dire ceux provoquant le plus d'attente aux points de synchronisation.

Annexe A

Programmes de test utilisés

Que l'on utilise un outil de calcul de temps d'exécution pire-cas par simulation ou par analyse statique, l'obtention de résultats nécessite toujours l'utilisation de programmes de test, ou *benchmarks*. Ces programmes permettent d'établir des valeurs-étalon à partir desquelles on va mesurer l'incidence, en nombre de cycles, de la modélisation de mécanismes internes au processeur (pipeline, cache ...) ou externes (bus, interruptions ...), et ainsi déterminer la fiabilité ou l'influence sur les performances de leur implémentation.

La plateforme OTAWA développée à l'IRIT au sein de l'équipe TRACES est un outil d'analyse statique de temps d'exécution pire-cas. Habituellement, les programmes de test utilisés proviennent des suites SNU-RT et Malardalen. Ce sont des petits programmes simples bien connus de la communauté temps-réel car ils ne réservent pas de mauvaise surprise au développeur. Nous avons également cherché à diversifier nos *benchmarks* en utilisant certaines fonctions des *MiBenchs* [1], des jeux de test initialement développés pour évaluer les performances des machines de calcul. Contrairement aux programmes habituels qui reposent principalement sur des fonctions de calcul, les *MiBenchs* couvrent une grande partie des applications actuelles des systèmes embarqués : automobile, réseaux, sécurité, multimédia, etc. Ce sont également des programmes relativement longs et qui donnent donc une meilleure idée des tâches pouvant être réellement exécutées sur un système temps-réel.

Le principal inconvénient des *MiBenchs* est qu'ils sont conçus pour être exécutés directement sur l'architecture cible, ou éventuellement par un simulateur. Le fait qu'ils ne soient pas ou peu utilisés dans les approches par analyse statique s'explique par certaines des caractéristiques intrinsèques du code, listées ci-dessous :

- l'analyse manuelle de flot (calcul des bornes de boucle) peut se révéler difficile voire impossible ;
- certains programmes utilisent l'allocation dynamique des emplacements mémoire, ce qui pose des problèmes lorsqu'on veut simuler la présence d'un cache de données ;
- les appels aux librairies système sont nombreux (affichage, allocation d'emplacements mémoire, etc.) or il n'est pas possible d'analyser ces appels ;
- de plus, certains programmes ne semblent pas tourner en code natif.

Par conséquent, ces fonctions ont dû subir quelques modifications afin d'être analysables par notre outil :

1. `http://www.eecs.umich.edu/mibench/`

```
inline float sqrtf(float number) {
    long i;
    float x, y;
    const float f = 1.5F;

    x = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;           // get bits for floating value
    i  = 0x5f3759df - ( i >> 1 );   // gives initial guess y0
    y  = * ( float * ) &i;          // convert bits back to float
        // Newton step, repeating increases accuracy
    y  = y * ( f - ( x * y * y ) );
        // Newton step, repeating increases accuracy
    y  = y * ( f - ( x * y * y ) );
    return number * y;
}
```

FIGURE A.1 – Fonction *racine carrée*

- nous n'utilisons que des fonctions suffisamment longues à exécuter (le temps d'exécution pire-cas se compte idéalement en millions de cycles) ;
- lorsque les fonctions font appels à des fonctions mathématiques de la librairie standard UNIX (`math.h`), nous avons remplacé ces fonctions par des fonctions produisant un résultat comparable mais dont le temps d'exécution peut être borné (un exemple est donné dans la figure A.1) ;
- les autres appels système sont soit commentés dans le code, soit dans le fichier d'analyse de flot (l'instruction `nocall` permet d'ignorer un branchement donné).

Le tableau A.1 liste les différents programmes de test utilisés dans les résultats expérimentaux des chapitres 3 et 4. Le tableau A.2 liste les temps d'exécution calculés par OTAWA en considérant une plateforme mono ou octo-cœurs et un cache de données parfait ou absent.

Pour les autres caractéristiques (nombre de requêtes sur le bus, ratio d'accès au bus, sensibilité du temps d'exécution pire-cas), se référer au tableau 3.3 en section 3.4.3.

Benchmark	Function
`nsichneu`	Simulation d'un réseau de Petri étendu. Code généré automatiquement et contenant beaucoup de structures conditionnelles (plus de 250).
`statemate`	Code généré automatiquement par l'outil STAtechart Real-time-Code generator STARC.
`compress`	Programme de compression de données provenant de la suite de *benchs* SPEC95. La compression est effectuée sur un tampon contenant une petite quantité de données aléatoires.
`susan_corners_quick`	Algorithme rapide de détection d'angles provenant de la suite SUSAN (programme de traitement d'image bas-niveau).
`susan_edges_small`	Algorithme rapide de détection de bords de SUSAN.
`susan_principle`	Algorithme d'application de filtre de SUSAN.
`edge_draw`	Algorithme de tracé de bords de SUSAN.
`corner_draw`	Algorithme de tracé de coins de SUSAN.
`setup_brightness_lut`	Fonction d'initialisation de la *look-up table* de luminosité dans SUSAN.
`susan_principle_small`	Algorithme rapide d'application de filtre de SUSAN.
`enlarge`	Algorithme d'élargissement d'image de SUSAN.
`susan_thin`	Algorithme de rétrécissement d'image de SUSAN.
`susan_corners`	Algorithme de détection d'angles de SUSAN.
`susan_edges`	Algorithme de détection de bords de SUSAN.
`matmult`	Calcul du produit de matrices.
`ns`	Recherche dans un tableau multi-dimensionnel.

TABLE A.1 – Programmes de test utilisés.

i	*Benchmark*	*Data Always-Hit* WCET		*Data Always-Miss* WCET	
		1-cœur	8-cœur	1-cœur	8-cœur
0	`nsichneu`	529.409	3.633.511	687.763	4.180.618
1	`statemate`	786.369	5.245.082	920.367	5.712.936
2	`susan_corners_quick`	2.125.304	12.220.502	5.016.023	33.738.971
3	`susan_edges_small`	2.260.287	8.250.051	7.186.993	46.021.703
4	`compress`	671.504	980.397	1.303.706	5.251.557
5	`susan_principle`	836.778	928.824	5.069.392	32.350.738
6	`edge_draw`	820.343	820.681	4.274.231	25.321.189
7	`corner_draw`	915.205	915.377	4.901.337	29.010.667

TABLE A.2 – Temps d'exécution pire-cas de certaines tâches du jeu de test utilisé. Taille du cache d'instruction : 2 Ko

Table des figures

1.1 Les différentes étapes du calcul de temps d'exécution pire-cas par analyse statique. 16
1.2 Code source et code assemblé de la fonction *somme* 17
1.3 Graphe de flot de contrôle de la fonction *somme* 18
1.4 Préfixe et suffixe d'un bloc de base. 20
1.5 Effets de recouvrement entre blocs de base. 22

2.1 Représentation schématique d'une plateforme multi-cœurs interconnectée par un bus partagé. 32
2.2 Diagramme temporel d'une opération de lecture en mémoire. 33
2.3 Représentation schématique d'une plateforme multi-cœurs interconnectée par un réseau multi-étages. 34
2.4 Représentation schématique d'une plateforme multi-cœurs interconnectée par un bus en anneau bi-directionnel. 36
2.5 Exemple d'arbitrage de bus avec un ordonnanceur à priorités fixes. 37
2.6 Exemple d'arbitrage de bus avec la politique *First-Come First-Served*. . . . 38
2.7 Exemple d'arbitrage de bus avec la politique du tourniquet. 40

3.1 Représentation schématique de l'architecture multi-cœurs cible. L'interconnexion est assurée par un bus partagé administré par un arbitre à deux niveaux. 46
3.2 Les deux mécanismes d'arbitrage proposés. 47
3.3 Exemple d'arbitrage de bus avec le protocole *Group Round Robin* (GRR-{1,3,4}). 48
3.4 Exemple de partage de bus entre trois groupes avec le protocole *Geometric Latencies*. 49
3.5 Automate fini représentant le comportement d'un arbitre utilisant le protocole *Geometric Latencies*. 53
3.6 Exemple d'arbitrage de bus avec le protocole *Geometric Group Latencies* (GGL-{1,3,4}). 53
3.7 Diagramme temporel d'une opération de lecture en mémoire. 56
3.8 Sensibilité du temps d'exécution pire-cas en fonction de la latence de bus et en considérant un cache de données absent (*Data Always-Miss*). 58
3.9 Sensibilité du temps d'exécution pire-cas en fonction de la latence de bus et en considérant un cache de données *parfait* (*Data Always-Hit*). 59

3.10 Nombre d'allocations possibles de tâches aux groupes $\gamma_{\{N_0,N_1,N_2\}}$ pour chaque configuration $\{N_0, N_1, N_2\}$. 62
3.11 Meilleures sommes des temps d'exécution pire-cas des tâches de test (en nombre de cycles) observés avec un cache de données parfait (*Always Hit*). 64
3.12 Meilleurs temps d'exécution pire-cas maximum de l'ensemble de tâches de test (en nombre de cycles) observés avec un cache de données parfait (*Always Hit*). 64
3.13 Meilleures sommes des temps d'exécution pire-cas des tâches de test (en nombre de cycles) observés sans cache de données (*Always Miss*). 66
3.14 Meilleurs temps d'exécution pire-cas maximum de l'ensemble de tâches de test (en nombre de cycles) observés sans cache de données (*Always Miss*). . 66

4.1 Taux d'utilisation global U avec l'algorithme de *bin-packing*. 86
4.2 Taux d'utilisation maximal par cœur $max(U_k)$ avec l'algorithme de *bin-packing*. 87
4.3 Évolution du taux d'utilisation global U, du nombre de tâches non ordonnançables et du nombre d'ensembles de tâches non ordonnançables en fonction du nombre d'itérations de l'algorithme 4.1 et en considérant la configuration GGL-{1,2,3}. 88
4.4 Nombre de configurations ayant au moins une solution ordonnançable et nombre moyen d'itérations de l'algorithme en fonction du taux d'utilisation par tâche u. 93
4.5 Taux d'utilisation global moyen observé U_{avg} et charge de référence en fonction du taux d'utilisation par tâche u. 94

A.1 Fonction *racine carrée* 102

Liste des tableaux

1.1 Système de contraintes généré par la méthode IPET. 19

3.1 Programmes de test utilisés. 55
3.2 Configuration des cœurs. 55
3.3 Critères des tâches du jeu de test utilisé (taille du cache d'instructions : 2 Ko) 57
3.4 Configurations de l'arbitre testées. 60
3.5 Meilleurs résultats de temps d'exécution pire-cas (en nombre de cycles) sans cache de données (*Data Always-Miss*). 63
3.6 Meilleurs résultats de temps d'exécution pire-cas (en nombre de cycles) avec un cache de données parfait (*Data Always-Hit*). 65
3.7 Priorisation des tâches en fonction des différents critères retenus avec un cache de données absent. 68
3.8 Temps d'exécution obtenus en fonction des différents critères d'allocation des tâches aux groupes avec un cache de données absent. 68
3.9 Priorisation des tâches en fonction des différents critères retenus avec un cache de données parfait. 69
3.10 Temps d'exécution obtenus en fonction des différents critères d'allocation des tâches aux groupes avec un cache de données parfait. 70

4.1 Solveurs ILP utilisés. 78
4.2 Quelques algorithmes d'ordonnancement de tâches généralement utilisés. . 83
4.3 Taux d'utilisation global minimum $min(U)$ obtenu pour chaque configuration avec l'algorithme 4.1 (sans prise en compte de la *whitelist*) et un taux d'utilisation par tâche u égal à 0,21. 89
4.4 Taux d'utilisation global minimum $min(U)$ obtenu pour chaque configuration avec l'algorithme 4.1 (avec prise en compte de la *whitelist*) et un taux d'utilisation par tâche u égal à 0,21. 92

A.1 Programmes de test utilisés. 103
A.2 Temps d'exécution pire-cas de certaines tâches du jeu de test utilisé. Taille du cache d'instruction : 2 Ko 103

Bibliographie

[1] H. Agrou, M. Gatti, P. Sainrat, and P. Toillon. A Design Approach for Predictable and Efficient Multi-Core Processor for Avionics. In *Digital Avionics Systems Conference*, volume 7D3, pages 1–11, 2011.

[2] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 3–14, 2008.

[3] H. A. Aljifri, A. Pons, and M. A. Tapia. Tighten the Computation of Worst-Case Execution Time by Detecting Feasible Paths. In *Proceedings of the IEEE International conference on Performance, Computing, and Communications*, pages 430–436, 2000.

[4] P. Altenbernd. On The False Path Problem in Hard Real-Time Programs. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 102 –107, 1996.

[5] ARM, AMBA Specification Rev 2.0, 1999. www.arm.com.

[6] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. *International Conference on VLSI Design*, pages 103–110, 2008.

[7] O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratchpad-Based Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 1 :6–26, November 2002.

[8] C. Ballabriga and H. Cassé. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *20th Euromicro Conference on Real-Time Systems*, pages 341–350, 2008.

[9] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *12th Euromicro Conference on Real-Time Systems*, pages 81–88, 2000.

[10] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. *23rd IEEE International Real-Time Systems Symposium*, pages 279–288, 2002.

[11] A. Betts and G. Bernat. Tree-Based WCET Analysis on Instrumentation Point Graphs. *9th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 558–565, 2006.

[12] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Accurate Analysis of Memory Latencies for WCET Estimation. *16th International Conference on Real-Time and Network Systems*, 2008.

[13] B. D. Bui, R. Pellizzoni, D. K. Chivukula, and M. Caccamo. Real-Time Communication for Multicore Systems with Multi-Domain Ring Buses. In *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 23 –32, 2010.

[14] A. Burchard, J. Liebeherr, Yingfeng Oh, and S.H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. *IEEE Transactions on Computers*, 44 :1429–1442, 1995.

[15] G. C. Buttazzo. Rate Monotonic vs. EDF : Judgment Day. *Real-Time Systems*, 29 :5–26, 2005.

[16] H. Cassé and C. Rochange. OTAWA, Open Tool for Adaptative WCET Analysis. In *Design, Automation and Test in Europe (poster session)*, 2007.

[17] F. J. Cazorla, A. Ramirez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernandez. QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 24(4) :24–31, 2004.

[18] R. Chapman, A. Burns, and A. Wellings. Combining Static Worst-Case Timing Analysis and Program Proof. *Real-Time Systems*, 11 :145–171, 1996.

[19] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering*, 15(10) :1261–1269, 1989.

[20] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2) :10–21, 2007.

[21] IBM Microelectronics, CoreConnect Bus Architecture, 1999. www.ibm.com.

[22] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. *2nd International Symposium on Programming*, 1976.

[23] P. Crowley. Worst-Case Execution Time Estimation for Hardware-Assisted Multithreaded Processors. *Proceedings of the 2nd Workshop on Network Processors*, pages 36–47, 2003.

[24] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997.

[25] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza (Burguière), J. Reineke, B. Triquet, and R. Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Proceedings of Embedded Real Time Software and Systems*, 2010.

[26] S. Davari and S. K. Dhall. An On Line Algorithm for Real-Time Tasks Allocation. In *IEEE Real-Time Systems Symposium*, pages 194–200, 1986.

[27] M. de Michiel, A. Bonenfant, C. Ballabriga, and H. Cassé. Partial Flow Analysis with oRange. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 479–482. Springer Berlin / Heidelberg, 2010.

[28] M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *IEEE*

International Conference on Embedded and Real-Time Computing Systems and Applications, pages 161–166, 2008.

[29] GOThA (Groupe de recherche en Ordonnancement Théorique et Appliqué). *Les Problèmes d'Ordonnancement*, volume 27. Revue française d'Automatique, d'Informatique et de Recherche Opérationnelle, 1995.

[30] J.-F. Deverge and I. Puaut. Safe Measurement-Based WCET Estimation. In *Proceedings of the 5th International Workshop on Worst Case Execution Time Analysis*, pages 13–16, 2005.

[31] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1) :127–140, 1978.

[32] S. A. Edwards and E. A. Lee. The Case for the Precision Timed (PRET) Machine. In *Proceedings of the 44th annual Design Automation Conference*, pages 264–265, 2007.

[33] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating Worst-Case Execution Times Analysis for Optimized Code. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 146–153, 1998.

[34] J. Engblom and B. Jonsson. Processor Pipelines and Their Properties for Static WCET Analysis. In *Embedded Software*, volume 2491 of *Lecture Notes in Computer Science*, pages 334–348. Springer Berlin / Heidelberg, 2002.

[35] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer Berlin / Heidelberg, 2001.

[36] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. *ACM SIGPLAN workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1997.

[37] C. Ferdinand and R. Wilhelm. On Predicting Data Cache Behavior for Real-Time Systems. In *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin / Heidelberg, 1998.

[38] C. Ferdinand and R. Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17 :131–181, 1999.

[39] R. Fuchsen. How to Address Certification for Multi-Core Based IMA Platforms : Current Status and Potential Solutions. In *29th IEEE/AIAA Digital Avionics Systems Conference*, pages 5.E.3–1 –5.E.3–11, 2010.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[41] J. Gustafsson and A. Ermedahl. Automatic Derivation of Path and Loop Annotations in Object-Oriented Real-Time Programs. *Journal of Parallel and Distributed Computing Practices 1(2)*, pages 61–74, 1998.

[42] J. Gustafsson and A. Ermedahl. Experiences from Applying WCET Analysis in Industrial Settings. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 382 –392, 2007.

[43] D. Hardy, T. Piquet, and I. Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *30th IEEE Real-Time Systems Symposium*, pages 68 –77, 2009.

[44] D. Hardy and I. Puaut. WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches. In *Real-Time Systems Symposium, 2008*, pages 456 –466, 2008.

[45] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting Timing Analysis by Automatic Bounding of Loop Iterations. *Real-Time Systems*, 18 :129–156, 2000.

[46] I. Hur and C. Lin. Adaptive History-Based Memory Schedulers. In *37th IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, 2004.

[47] K. Jeffay and D. F. Stanat. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 129–139, 1991.

[48] D. S. Johnson. Fast Algorithms for Bin-Packing. *Journal of Computer and System Sciences*, 8(3) :272–314, 1974.

[49] D. Kanter. The Common System Interface : Intel's Future Interconnect, 2007. http ://www.realworldtech.com/page.cfm ?ArticleID=rwt082807020032.

[50] S.-K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Real-Time Technology and Applications Symposium*, pages 230–240, 1996.

[51] L. Ko, D. B. Whalley, and M. G. Harmon. Supporting User-Friendly Analysis of Timing Constraints. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 99–107, 1995.

[52] A. A. Kountouris. Safe and Efficient Elimination of Infeasible Execution Paths in WCET Estimation. In *Proceedings of the 3rd International Workshop on Real-Time Computing Systems and Applications*, pages 187 –194, 1996.

[53] Kevin Krewell. Multicore Showdown. *Microprocessor Report*, 19 :41–45, 2005.

[54] R. Kumar, V. Zyuban, and D.M. Tullsen. Interconnections in Multi-Core Architectures : Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 408–419, 2005.

[55] C. C. Lee and D. T. Lee. A Simple On-Line Bin-Packing Algorithm. *Journal of the Association for Computing Machinery*, 32 :562–572, 1985.

[56] J. P. Lehoczky and L. Sha. Performance of Real-Time Bus Scheduling Algorithms. In *Proceedings of the ACM SIGMETRICS joint international conference on Computer Performance Modelling, Measurement and Evaluation*, pages 44–53, 1986.

[57] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *30th IEEE Real-Time Systems Symposium*, pages 57–67, 2009.

[58] Y.-T. Li and S. Malik. Peformance Analysis of Software Using Implicit Path Enumeration. *16th IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1997.

[59] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *Workshop on Languages, Compilers, and Tools for Real-time Systems*, 1995.

[60] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software : Beyond Direct Mapped Instruction Caches. In *17th IEEE Real-Time Systems Symposium*, pages 254–263, 1996.

[61] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4 :257–279, 1999.

[62] M. Lindgren, H. Hansson, and H. Thane. Using Measurements to Derive the Worst-Case Execution Time. *7th International Workshop on Real-Time Computing Systems and Applications*, 0 :15, 2000.

[63] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 20 :46–61, 1973.

[64] Y. Liu and G. Gomez. Automatic Accurate Time-Bound Analysis for High-Level Languages. In *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 31–40. Springer Berlin / Heidelberg, 1998.

[65] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Processors with Cache Memories*. PhD thesis, Chalmers University of Technology, 2002.

[66] D. Macos and F. Mueller. Integrating GNAT/GCC Into a Timing Analysis Environment. In *Proceedings of the 10th EUROMICRO Workshop on Real-Time Systems*, page 15–18, 1998.

[67] P. McMinn. Search-Based Software Test Data Generation : A Survey. *Software Testing, Verification and Reliability*, 14(2) :105–156, 2004.

[68] F. Mueller. Timing Predictions for Multi-Level Caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, 1997.

[69] F. Mueller and D. Whalley. Efficient On-the-Fly Analysis of Program Behavior and Static Cache Simulation. In *Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 101–115. Springer Berlin / Heidelberg, 1994.

[70] F. Mueller and D. Whalley. Fast Instruction Cache Analysis via Static Cache Simulation. *Proceedings of the 28th Annual Simulation Symposium*, pages 105–114, 1995.

[71] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160, 2007.

[72] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, 2006.

[73] Y. Oh and S. H. Son. Allocating Fixed-Priority Periodic Tasks on Multiprocessor Systems. *Real-Time Systems*, 9 :207–239, 1995.

[74] M. Paolieri, E. Quinones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 57–68, 2009.

[75] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5 :31–62, 1993.

[76] S. M. Petters. Comparison of Trace Generation Methods for Measurement-Based WCET Analysis. In *Proceedings of the 3rd International Workshop on Worst Case Execution Time Analysis*, pages 61–64, 2003.

[77] S. Plazar, P. Lokuciejewski, and P. Marwedel. WCET-Aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In *9th International Workshop on Worst-Case Execution Time Analysis*, 2009.

[78] P.J. Prisaznuk. Integrated Modular Avionics. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, volume 1, pages 39–45, 1992.

[79] I. Puaut and D. Decotigny. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *23rd IEEE Real-Time Systems Symposium*, pages 114–123, 2002.

[80] P. Puschner. A Tool for High-Level Language Analysis of Worst-Case Execution Times. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 130–137, 1998.

[81] P. Puschner and A. Burns. Writing Temporally Predictable Code. In *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, 2002.

[82] P. Puschner and C. Koza. Calculating The Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1 :159–176, 1989.

[83] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, 2000.

[84] C. Rochange and P. Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Transactions on HiPEAC*, 2(3), 2007.

[85] E. Rohou, F. Bodin, A. Seznec, G. Le Fol, F. Charot, and F. Raimbault. SALTO : System for Assembly-Language Transformation and Optimization. Technical report, INRIA, 1996.

[86] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 49–60, 2007.

[87] A. C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15 :875–889, 1989.

[88] A. L. Shimpi. Intel's Sandy Bridge Architecture Exposed, 2010. http : //www.anandtech.com/show/3922/intels-sandy-bridgearchitecture-exposed.

[89] J. E. Smith and G. S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83(12) :1609–1624, 1995.

[90] Brinkley Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, 22 :64–71, 2002.

[91] J. A. Stankovic. Misconceptions about Real-Time Computing : A Serious Problem for Next-Generation Systems. *Computer*, 21(10) :10–19, 1988.

[92] J. Staschulat and M. Bekooij. Dataflow Models For Shared Memory Access Latency Analysis. *Proceedings of the 7th ACM International Conference on Embedded software*, pages 275–284, 2009.

[93] V. Suhendra and T. Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303, 2008.

[94] R. Słowiński. Scheduling Preemptable Tasks on Unrelated Processors with Additional Resources to Minimize Schedule Length. In *Information Systems Methodology*, volume 65 of *Lecture Notes in Computer Science*, pages 536–547. Springer Berlin / Heidelberg, 1978.

[95] K. C. Tan, T. H. Lee, and E. F. Khor. Evolutionary Algorithms for Multi-Objective Optimization : Performance Assessments and Comparisons. *Artificial Intelligence Review*, 17 :251–290, 2002.

[96] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems*, 18 :157–179, 2000.

[97] N. Tracey, J. Clark, and K. Mander. The Way Forward for Unifying Dynamic Test-Case Generation : The Optimisation-Based Approach. *International Workshop on Dependable Computing and Its Applications*, pages 169–180, 1998.

[98] D. Triolet. ARM Cortex A7 et big.LITTLE : le silicium noir, 2011. http : //www.hardware.fr/focus/52/arm-cortex-a7-big-little-silicium-noir.html.

[99] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading : Maximizing On-Chip Parallelism. In *Proceedings of the 22nd annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[100] M. K. Vernon and U. Manber. Distributed Round-Robin and First-Come First-Serve Protocols and their Applications to Multiprocessor Bus Arbitration. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 269–279, 1988.

[101] E. Wandeler and L. Thiele. Optimal TDMA time slot and cycle length allocation for hard real-time systems. *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 479–484, 2006.

[102] A. Watkins and E. M. Hufnagel. Evolutionary Test Data Generation : A Comparison of Fitness Functions. *Software : Practice and Experience*, 36(1) :95–116, 2006.

[103] J. Wegener and M. Grochtmann. Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. *Real-Time Systems*, 15 :275–298, 1998.

[104] J. Wegener and F. Mueller. A Comparison of Static Analysis and Evolutionary Testing for the Verification of Time Constraints. *Real-Time Systems*, 21 :241–268, 2001.

[105] N. Williams. WCET Measurment Using Modified Path Testing. In *5th International Workshop on Worst-Case Execution Time Analysis*, Dagstuhl, Germany, 2007. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[106] N. Williams, B. Marre, P. Mouy, and M. Roger. Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. *Dependable Computing - EDCC 5*, pages 281–292, 2005.

[107] A. W. Wilson Jr. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th annual International Symposium on Computer Architecture*, pages 244–252, 1987.

[108] J. Yan and W. Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.

[109] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, pages 214–225, 1990.

Résumé

Les défaillances des applications embarquées dans les systèmes temps-réel strict peuvent avoir des conséquences graves (catastrophes industrielles, mise en danger de vies humaines). La vérification des contraintes temporelles d'un système temps-réel strict dépend de la connaissance du temps d'exécution pire-cas des tâches constituant l'application embarquée. L'utilisation de processeurs multi-cœurs est l'un des moyens actuellement mis en œuvre afin d'améliorer le niveau de performances des systèmes embarqués. Cependant, la détermination du temps d'exécution pire-cas d'une tâche sur ce type d'architecture est rendue difficile par le partage de certaines ressources par les cœurs, et notamment le bus d'interconnexion permettant l'accès à la mémoire centrale. Ce document propose un nouveau mécanisme d'arbitrage de bus à deux niveaux permettant d'améliorer les performances des ensembles de tâches exécutés tout en garantissant le respect des contraintes temporelles. Les méthodes décrites permettent d'établir un niveau de priorité d'accès au bus optimal pour chacune des tâches exécutées. Elles permettent également de trouver une allocation optimale des tâches aux cœurs lorsqu'il y a plus de tâches à exécuter que de cœurs disponibles. Les résultats expérimentaux montrent une diminution significative des estimations de temps d'exécution pire-cas et de l'utilisation du processeur.

Abstract

Software failures in hard real-time systems may have hazardous effects (industrial disasters, human lives endangering). The verification of timing constraints in a hard real-time system depends on the knowledge of the worst-case execution times (WCET) of the tasks accounting for the embedded program. Using multicore processors is a mean to improve embedded systems performances. However, determining worst-case execution times estimates on these architectures is made difficult by the sharing of some resources among cores, especially the interconnection bus that enables accesses to the shared memory. This document proposes a two-level arbitration scheme that makes it possible to improve executed tasks performances while complying with timing constraints. Described methods assess an optimal bus access priority level to each of the tasks. They also allow to find an optimal allocation of tasks to cores when tasks to execute are more numerous than available cores. Experimental results show a meaningful drop in worst-case execution times estimates and processor utilization.