THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Antoine BLIN

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Vers une utilisation efficace des processeurs multi-cœurs dans des systèmes embarqués à criticités multiples.

Soutenue le 30 janvier devant le jury composé de :

Mme Claire PAGETTI	*Rapporteure*	Ingénieur de recherche (HDR), ONERA
Mme Isabelle PUAUT	*Rapporteure*	Professeur, université de Rennes 1/IRISA
Mme Béatrice BÉRARD	*Examinatrice*	Professeur, université Pierre et Marie Curie
M. Marc GATTI	*Examinateur*	Directeur recherche & technologie, Thales
Mme Sophie QUINTON	*Examinatrice*	Chargée de recherche, Inria
M. Gilles MULLER	*Directeur de thèse*	Directeur de recherche, Inria
M. Julien SOPENA	*Encadrant*	Maître de conférences, université Pierre et Marie Curie
M. Philippe QUERE	*Invité*	Gestionnaire de projet Alliance, Renault

Remerciements

Je souhaite ici exprimer ma gratitude envers les nombreuses personnes qui m'ont soutenu, encouragé et motivé tout au long de cette thèse.

Un grand merci à Claire Pagetti et à Isabelle Puaut, qui ont rapporté cette thèse, pour le temps qu'elles ont consacré à la relecture de ce manuscrit et pour leurs commentaires et leurs avis. Je remercie, en outre, Béatrice Bérard, Marc Gatti, Sophie Quinton, Philippe Quere pour l'attention qu'ils ont porté à mon travail en acceptant de participer à mon jury.

Cette thèse n'aurait jamais pu arriver à son terme sans les précieux conseils et les encouragements de mes encadrants, Gilles Muller et Julien Sopena, ainsi que Julia Lawall qui, par leur disponibilité et leur compétence, m'ont soutenu, guidé et conseillé tout au long du déroulement de mes travaux et de la rédaction de ce manuscrit. Merci de m'avoir permis de m'investir dans cette problématique captivante. J'ai, durant cette thèse, pu animer des stages et entretenir ainsi une collaboration des plus fructueuses qui m'a permis d'avancer dans mes travaux. Un grand merci à Cédric, et Redha : j'ai particulièrement apprécié de travailler avec vous.

Je remercie mes encadrants industriels Youssef Laarouchi, Philippe Querre et Rémy Brugnon, pour avoir rendu cette thèse possible et m'avoir suivi durant ces trois ans, ainsi que mes collègues de Renault dont les conseils et l'expertise industrielle a été particulièrement précieuse : Joris, Régis, Patrick. Merci également à tous les doctorants rencontrés chez Renault, Sylvain, Ivan, Hélène, pour leur soutien sans faille. Je remercie l'équipe de l'IRT SystemX au sein de laquelle j'ai pu avoir de fructueux échanges.

Je remercie également tous les membres des équipes WHISPER, REGAL et MOVE -notamment Sébastien, Pierre-Evariste, Pierre, Luciana, Swan, Yann, Quentin et tant d'autres- au sein desquelles j'ai pu m'épanouir durant la durée de ma thèse. Merci à tous les doctorants que j'ai pu rencontrer durant ces années et avec qui j'ai passé de très bon moments Denis, Marjorie, Gauthier, Maxime, Ilyas, Damien, Rudyar, Lyes, Hakan, Ludovic, Karine, Hamza, Maxime.

Je n'aurai pas pu faire cette thèse sans le soutien de mes amis, qui m'ont encouragé durant ces quatre longues années. Merci à Olivier, Maxime et Lolo, pour tous les moments que nous avons passés ensemble aussi bien au ski, au tennis, qu'à la piscine, dans les bars et à l'escalade. Vous me manquez beaucoup et je pense très souvent à vous.

Merci à ma famille qui m'a soutenu tout au long de mes études et qui m'a accompagné jusque-là. Je pense très fort à mes parents ainsi qu'a ma sœur Fofie, dont les encouragements et le soutien ont largement contribué à l'accomplissement de mon travail. Des bisous à mes oncles et tantes et à tous mes cousins et plein de poutounes pour la petite Justine qui vient de nous rejoindre.

Table des matières

1 Contexte industriel automobile **5**
- 1.1 Systèmes du monde automobile . . . 6
  - 1.1.1 Architecture Électrique/Électronique . . . 7
  - 1.1.2 Architecture des systèmes de divertissement du consommateur . . . 10
- 1.2 Tendances, limitations et solutions . . . 13
  - 1.2.1 Tendances . . . 13
  - 1.2.2 Limitations . . . 15
  - 1.2.3 Solutions : virtualisation sur des systèmes multi-cœurs . . . 17
- 1.3 Conclusion . . . 18

2 Introduction au problème de contention mémoire **21**
- 2.1 Virtualisation . . . 22
  - 2.1.1 Définitions et concepts . . . 22
  - 2.1.2 Virtualisation des composants . . . 23
  - 2.1.3 Virtualisation des systèmes embarqués . . . 29
- 2.2 Multi-cœurs . . . 30
  - 2.2.1 Modèle de programmation multi-cœurs sur architectures virtualisées . . . 30
  - 2.2.2 Multi-cœurs et contraintes temps réel . . . 31
- 2.3 Hiérarchie mémoire . . . 33
  - 2.3.1 Caractéristiques des mémoires . . . 34
  - 2.3.2 Caches . . . 35
  - 2.3.3 Mémoire principale . . . 39

3 Etat de l'art **45**
- 3.1 Approche par composants matériels . . . 46
  - 3.1.1 Classification des solutions . . . 46
  - 3.1.2 Caches processeurs . . . 47
  - 3.1.3 Contrôleur mémoire . . . 55
  - 3.1.4 Mémoire principale . . . 58
  - 3.1.5 Bus matériels . . . 61
  - 3.1.6 Conclusion . . . 63

3.2 Approches globales . . . 63
3.2.1 Contrôle des accès mémoire . . . 64
3.2.2 Régulation des accès mémoire . . . 69
3.3 Conclusion . . . 73

4 Mise en évidence expérimentale du problème 75
4.1 Plate-forme matérielle . . . 76
4.1.1 Processeur . . . 77
4.1.2 Hiérarchie mémoire de niveau 1 . . . 79
4.1.3 Cache L2 . . . 80
4.1.4 Contrôleur mémoire . . . 83
4.1.5 Récapitulatif . . . 85
4.2 Stratégie de partage des ressources matérielles . . . 85
4.2.1 Partage des ressources CPU . . . 85
4.2.2 Partage des périphériques . . . 86
4.2.3 Partage de la hiérarchie mémoire . . . 87
4.3 Méthode d'évaluation . . . 88
4.3.1 Configuration du matériel . . . 89
4.3.2 Linux . . . 89
4.3.3 Benchmark : MiBench . . . 90
4.3.4 Charges . . . 92
4.4 Évaluation . . . 96
4.4.1 Coût du partage du cache L2 . . . 96
4.4.2 Impact de la contention mémoire sur les temps d'exécutions . . . 97
4.5 Conclusion . . . 97

5 Une nouvelle approche de contrôle de la mémoire 99
5.1 Approche . . . 100
5.1.1 Étape hors ligne : caractérisation de la plate-forme . . . 101
5.1.2 Étape en ligne : contrôle . . . 102
5.2 Caractérisation de la plate-forme logicielle et matérielle . . . 103
5.2.1 Phases applicatives . . . 104
5.2.2 Génération de charges mémoires constantes paramétrables . . . 106
5.2.3 Collecte des données . . . 109
5.2.4 Traitement des données et génération des tables . . . 111
5.3 Contrôle à l'exécution . . . 114
5.3.1 Algorithme . . . 115
5.3.2 Mesure de la consommation mémoire . . . 116
5.3.3 Surveillance par échantillonnage . . . 116
5.3.4 Tables embarquées . . . 117
5.3.5 Suspension des applications . . . 118
5.4 Evaluation . . . 118
5.4.1 Surcoût à l'exécution . . . 119
5.4.2 Efficacité pour des charges constantes . . . 120

5.5 Conclusion 127

6 Analyse des causes de la sensibilité mémoire des applications temps réel 129
6.1 Profilage des applications 130
6.1.1 Profileur mémoire 130
6.1.2 Classification 134
6.1.3 Méthodologie d'investigation des pics de bande passante mémoire . . 138
6.2 Impacts des entrée/sortie 139
6.2.1 Analyse du problème 139
6.2.2 Conclusion & solutions 144
6.3 Impacts du système d'exploitation 145
6.3.1 Analyse du problème 145
6.3.2 Conclusion & solutions 146
6.4 Impacts des algorithmes applicatifs 146
6.4.1 Analyse du problème 146
6.4.2 Conclusion & solutions 148
6.5 Conclusion 148

Bibliographie 157

Table des figures

1.1 Architectures 6

2.1 Types d'hyperviseurs 23
2.2 Mémoire paginée 26
2.3 Partage des droits mémoire 27
2.4 Modèle de programmation multi-cœurs 30
2.5 Cas d'utilisation AMP 32
2.6 Hiérarchie mémoire 35
2.7 Placement des données 36
2.8 Terminologie des caches 38
2.9 DRAM 41
2.10 Latences d'accès à la mémoire 41
2.11 Politiques de gestion des *row-buffers* 43

3.1 Exemple de modèle d'exécution déterministe 65
3.2 PikeOS 67
3.3 Marthy 68
3.4 Memguard 70
3.5 Contrôle à l'exécution par mesure des anomalies temporelles 73

4.1 Architecture matérielle simplifiée de la carte SABRE Lite 76
4.2 Système monoprocesseur Cortex-A9 [10] 77
4.3 Exemple de configuration multiprocesseur [8] 78
4.4 Exemple de contrôleur de cache interfacé avec un processeur ARM [117] . . . 82
4.5 Diagramme de bloc du contrôleur MMDC [118] 83
4.6 Boucle principale du programme « charge » développé pour solliciter la hiérarchie mémoire 95
4.7 Impact du partitionnement du cache L2 sur les performances des appréciations issues de la suite de test MiBench. 96
4.8 Impact des applications « charges » sur les performances des appréciations issues de la suite de test MiBench avec différentes configurations de partitionnement du cache L2. 98

5.1 Étape hors ligne . . . 101
5.2 Étape en ligne . . . 103
5.4 Profils mémoire des applications ayant des phases applicatives sélectionnées au sein de la suite MiBench . . . 105
5.5 Boucle principale du programme charge modifié pour solliciter la hiérarchie mémoire avec des bandes passantes et des taux de lecture et d'écriture variables 107
5.6 Bande passante totale d'une charge exécutée en isolation . . . 109
5.7 Bande passante en lecture d'une charge exécutée en isolation . . . 109
5.8 Bande passante en écriture d'une charge exécutée en isolation . . . 109
5.9 Tables des ralentissements des quatre phases de `susan small -c` . . . 114
5.10 Ralentissement des applications sélectionnées au sein de la suite MiBench . . . 122
5.14 Parallélisme . . . 127

6.1 Bande passante mémoire générée par le profileur . . . 133
6.2 Profils mémoire des applications de MiBench exécutées avec différents intervalles d'échantillonnage. . . . 135
6.2 Profils mémoire des applications de MiBench exécutées avec différents intervalles d'échantillonnage. . . . 136
6.2 Profils mémoire des applications de MiBench exécutées avec différents intervalles d'échantillonnage. . . . 137
6.3 Gros plan sur une partie du profil tagué de l'application `Patricia small` . . . 139
6.4 Profils mémoire de `Rijndael small encode` exécutée avec les appels à la fonction `fwrite` inhibés et en faisant varier les tailles du tampon de cache pour le flux des données d'entrée de l'application . . . 141
6.5 Profil mémoire de `Rijndael small encode` exécutée avec les appels à la fonction `fwrite` inhibés et avec une fonction de lecture, des données d'entrée de l'application, « bas niveau » . . . 142
6.6 Profil mémoire de `Rijndael small encode` exécutée avec une fonction de lecture, des données d'entrée de l'application, « bas niveau » en faisant varier les tailles de tampon pour le flux de sortie . . . 143
6.7 Profils mémoire d'`ADPCM small` avec les écritures commentées et une résolution d'1 µs . . . 143
6.8 Profils mémoire d'`ADPCM small` effectués avec les écritures commentées et un petit tampon de flux de lecture avec une résolution d'1 µs . . . 144
6.9 Profils mémoire de plusieurs exécutions de l'application à faible empreinte mémoire . . . 145
6.10 Profil mémoire de l'application à faible empreinte mémoire avec un noyau dont l'horloge est cadencée à 50 Hz . . . 146
6.11 Profils mémoire d'étude de l'application `Susan large -c` . . . 147
6.12 Solutions futures proposées . . . 153

Liste des tableaux

2.1 Effets indésirables affectant le déterminisme temporel des composants partagés dans les architectures multi-cœurs [74] 33

3.1 Synthèse des solutions de l'état de l'art classifiées par famille d'approche . . . 47
3.2 Tableau comparatif des solutions de partitionnement *index-based* multi-cœurs utilisant des implémentations matérielles [52]. 50
3.3 Tableau comparatif des solutions de partitionnement *index-based* multi-cœurs utilisant des implémentations logicielles [52]. 52
3.4 Tableau comparatif des solutions de partitionnement *way-based* multi-cœurs n'utilisant pas de *cache-locking* [52] 53
3.5 Tableau comparatif des solutions de *cache-locking* s'appliquant aux multi-cœurs 54

4.1 Caractéristiques matérielles des différents niveaux de la hiérarchie mémoire. . 85
4.2 Applications sélectionnées 91
4.3 Temps d'exécution des applications MiBench exécutées avec le cache non partitionné 93

5.1 Ralentissements des appplications de la suite MiBench provoqué par les interruptions 120

6.1 Fonctions d'entrée/sortie sources de pics de bande passante mémoire 140

Introduction

Le logiciel a pris, dans les véhicules modernes, une place de plus en plus importante, tant dans les phases de conception et de développement, que dans les phases de commercialisation du produit.

Historiquement, une distinction a été effectuée entre les systèmes dits **mécatroniques** développés pour piloter les organes mécaniques du véhicule et les systèmes informatiques de divertissement du consommateur (radio puis navigation GPS et enfin multimédia). Les systèmes mécatroniques doivent intégrer dans leur cycle de développement des contraintes relevant de la sûreté de fonctionnement, du temps réel et de l'embarquabilité tandis que le second type de systèmes est sujet à des exigences moins rigoureuses.

La faible puissance de calcul des premières unités de contrôles électroniques utilisées dans les systèmes mécatroniques a conduit à la mise en place d'une architecture logicielle et matérielle dite **fédérée**. Chaque fonctionnalité y est implantée sous forme de logiciel s'exécutant sur des unités de calculs dédiées, l'ensemble de ces unités étant connectées entre elles par un ou plusieurs réseaux en bus (CAN, LIN, FLEXRAY...). Ce type d'architecture facilite la conception en termes de **sûreté de fonctionnement**, puisqu'il implique une séparation physique entre les différentes fonctionnalités. Seul le réseau peut servir de canal de propagation d'erreurs. En revanche, chaque introduction d'une nouvelle prestation client dans le véhicule entraîne l'ajout d'une ou de plusieurs unités de calculs. Or, l'augmentation du nombre d'équipements pose des problèmes en termes de consommation électrique, de dissipation thermique, de volume physique pris dans l'habitacle et d'infrastructure réseau entraînant, par conséquent, une augmentation des coûts.

L'émergence de calculateurs plus puissants a permis le passage d'une architecture fédérée à une architecture dite **intégrée** dans laquelle plusieurs fonctionnalités sont regroupées au sein d'une même unité de calcul. Le standard AUTOSAR, élaboré par des acteurs du monde automobile, a pour but de normaliser le développement logiciel sur de telles architectures, pour en favoriser la portabilité, la modularité et la réutilisation. Les bénéfices de cette architecture se traduisent par une réduction du poids et du volume pris par l'électronique au sein du véhicule ainsi que par la baisse de la consommation énergétique et de la dissipation thermique. En revanche, l'architecture intégrée abolit la séparation physique qui existait entre les composants logiciels ce qui entraîne des difficultés dans la mise en œuvre de la sûreté de fonctionnement.

Historiquement, un compromis avait été trouvé en séparant les systèmes de divertissement du consommateur (musique puis GPS/navigation et enfin vidéos et jeux) des systèmes mécatroniques, dans une architecture hybride intégrée-fédérée. Ainsi, les deux systèmes y étaient déployés sur des unités de calculs distinctes reliées au travers d'une passerelle assurant le filtrage des communications inter-calculateurs.

Le développement des calculateurs multi-cœurs achetés sur étagères (*Commercial Off-The-Shelf*) plus puissants permet d'envisager une nouvelle étape quant à l'intégration des systèmes multimédia et des systèmes mécatroniques : l'architecture **virtualisée**. Il s'agit de faire cohabiter sur un même calculateur une ou plusieurs instances des systèmes d'exploitation mécatroniques (Normes OSEK/VDX, AUTOSAR) et des systèmes d'exploitation multimédia (Android, Norme GENIVI) en utilisant une couche logicielle nommée hyperviseur qui gère l'accès au matériel. La virtualisation offre d'importantes avancées en termes d'intégration, la co-localité physique des systèmes facilitant grandement le partage des composants matériels. Néanmoins, l'architecture virtualisée abolit la séparation physique existante entre les différents systèmes. L'hyperviseur doit donc prendre le relais, pour assurer un partage du matériel et l'isolation entre les systèmes afin de garantir le respect des contraintes de sûreté et de sécurité.

Or, s'il est possible de mettre en place des mécanismes permettant d'assurer l'isolation, lors du partage de ressources matérielles (CPU, périphériques, ...), cela reste particulièrement délicat pour la mémoire. En effet, s'il reste aisé de la partitionner spatialement (MMU, MPU), cela n'empêche pas les problèmes de contention sur le bus mémoire et les interconnects. Ainsi, des logiciels non temps réel qui effectueraient un grand nombre d'accès à la mémoire peuvent ralentir un logiciel temps réel exécuté en parallèle sur un cœur dédié conduisant, par la même, à une violation de ses échéances. La solution de simplicité, pour supprimer ces interférences, réside dans un partage temporel strict : lorsque une application temps réel s'exécute, les autres cœurs sont désactivés. Une telle approche présente l'inconvénient majeur de diminuer l'intérêt des plates-formes multi-cœurs. Le but de ma thèse CIFRE, effectuée en collaboration avec Renault, est de proposer une nouvelle approche permettant d'assurer les propriétés temps réel tout en maximisant le parallélisme. L'idée est de retarder, le plus possible, la coupure des cœurs *best effort*, voire de maintenir un parallélisme total lorsque cela ne pose pas de problèmes. Si des approches en ce sens existent déjà, l'originalité de mes travaux réside dans la prise en compte de contraintes industrielles fortes : pas de modifications des applications, documentation partielle limitant les approches par calculs de WCET et carte matérielle à bas coûts, offrant peu de compteurs matériels.

En préambule de ces travaux, nous avons commencé par démontrer et quantifier le problème d'isolation temporelle sur une carte multi-cœurs embarquée COTS i.MX 6 SABRE Lite largement utilisée dans le monde automobile. Pour ce faire, nous avons développé une plateforme expérimentale basée sur des charges logicielles paramétrables permettant de stresser le bus mémoire. Nous y avons aussi intégré un ensemble de capteurs afin de mesurer l'effet de la contention sur des applications temps réel, issues de la suite de test *MiBench*. Les résultats de ces expériences ont montré que le problème était significatif, la contention pouvant générer des retards allant jusqu'à 183% du temps nominal.

Forts de ces résultats et en s'appuyant sur un état de l'art des solutions existantes, nous proposons ensuite une nouvelle approche logicielle de régulation des accès mémoire qui s'ar-

ticule autour de deux étapes. Tout d'abord, nous proposons une méthode de profilage hors ligne de l'application temps réel. Le but est de caractériser la sensibilité à la contention des différentes phases de l'application temps réel sous forme d'une table associant à une bande passante mesurée un retard estimé. Dans une deuxième étape, en ligne, nous utilisons cette table dans un oracle permettant de détecter les éventuels ralentissements des tâches temps réel et, le cas échéant, de suspendre l'exécution des applications *best effort*. Une évaluation de performances a montré non seulement que le surcoût dû au mécanisme de contrôle était minime, mais aussi qu'il permettait de garantir le respect des échéances tout en maintenant un bon niveau de parallélisme.

Dans un troisième temps, fort de l'expérience acquise lors de la conception de notre mécanisme et, en nous appuyant sur notre plate-forme expérimentale, nous avons cherché à comprendre l'origine de la sensibilité des applications temps réel. Les résultats de cette étude nous ont permis d'en dégager trois causes : les Entrées/Sorties, le système d'exploitation et les algorithmes applicatifs. Nous avons ainsi pu proposer des pistes d'améliorations pour désensibiliser les applications temps réel.

Le premier chapitre de ce manuscrit est consacré à la présentation du contexte industriel dans lequel s'inscrivent nos travaux. Nous effectuerons une rétrospective de l'architecture matérielle et logicielle du monde automobile. Puis, nous détaillons les futures tendances quant à l'évolution de cette architecture.

Le deuxième chapitre présente le contexte technique nécessaire à la mise en œuvre d'une architecture virtualisée sur des processeurs multi-cœurs. Nous détaillerons les principes de la virtualisation pour ensuite aborder les problèmes d'isolation temporelle posés par l'utilisation d'architectures multi-cœurs COTS.

Dans le troisième chapitre, nous effectuerons un état de l'art des mécanismes matériels et logiciels d'isolation temporelle du système mémoire pour des architectures multi-cœurs.

Le quatrième chapitre est consacré à la mise en évidence du problème d'isolation temporelle au sein d'une carte embarquée utilisée dans le monde automobile.

Dans le cinquième chapitre, nous détaillons l'approche de régulation que nous avons proposé pour gérer les problèmes de contention mémoire.

Enfin, dans le sixième et dernier chapitre de notre manuscrit, nous effectuerons une étude de la consommation mémoire des applications temps réel MiBench utilisées comme tests de référence.

Nous concluons ce manuscrit par une synthèse des travaux réalisés et par une présentation des perspectives de travaux futurs.

Chapitre 1

Contexte industriel automobile

Sommaire

1.1 Systèmes du monde automobile 6
1.1.1 Architecture Électrique/Électronique 7
1.1.2 Architecture des systèmes de divertissement du consommateur 10
1.2 Tendances, limitations et solutions 13
1.2.1 Tendances 13
1.2.2 Limitations 15
1.2.3 Solutions : virtualisation sur des systèmes multi-cœurs 17
1.3 Conclusion 18

L'élaboration d'un projet automobile fait l'objet d'un processus d'ingénierie complexe. L'étape de rédaction du cahier des charges a pour but de formaliser les prestations devant être implantées dans le véhicule. Au cours des dernières décennies, de plus en plus de prestations (Contrôle du moteur, Climatisation, ABS, ...) ont été réalisées en utilisant des composants électroniques pilotés par des programmes logiciels. Pour faire face à la complexité accrue des développements véhicules, induite par l'utilisation de ces nouvelles technologies, des approches [33, 70, 113] d'ingénierie logicielle ont été proposées. La figure 1.1 illustre le découpage en quatre niveaux d'architecture qui est préconisé par la démarche d'ingénierie afin de découpler les spécifications fonctionnelles des implantations logicielles et matérielles .

L'**architecture fonctionnelle** a pour objectif de formaliser et de décrire, à haut niveau, les services définis dans le cahier des charges. Chaque service est découpé en une hiérarchie de blocs fonctionnels connectés en réseau où chaque bloc spécifie une fonctionnalité attendue en s'abstrayant des détails de fonctionnement interne.

L'**architecture logicielle** décrit comment les fonctionnalités de l'architecture fonctionnelle sont implantées dans le logiciel. Elle spécifie le découpage des différents blocs fonctionnels en composants logiciels.

L'**architecture matérielle** détaille l'ensemble des composants matériels présents dans le

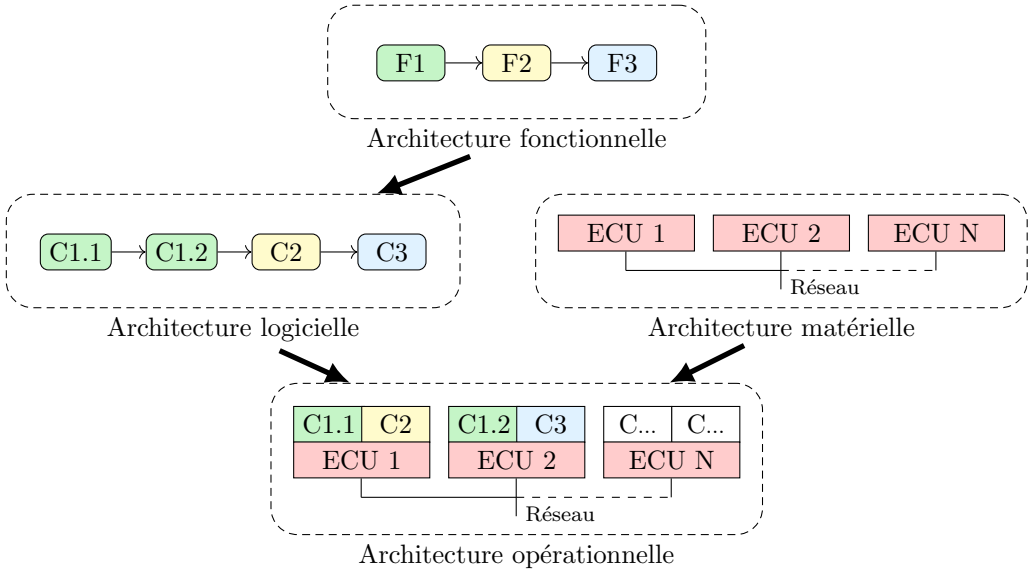


FIGURE 1.1 – Architectures

véhicule. Elle comprend les calculateurs, nommés aussi unités de contrôle électronique (*Electronic Control Unit*) reliés entre eux par des bus réseaux (CAN [137], LIN [138], FlexRay [139]). Les ECUs recueillent des informations sur l'environnement du véhicule en utilisant des capteurs et les transmettent à l'applicatif qui agit sur les organes mécaniques et hydrauliques du véhicule en utilisant des actionneurs.

L'**architecture opérationnelle** décrit l'association faite entre l'**architecture logicielle** (les composants logiciels) et l'**architecture matérielle** (les ECUs).

Dans ce chapitre, nous présentons les systèmes embarqués dans l'automobile et les problématiques qui y sont associées. Tout d'abord, nous détaillons les caractéristiques fonctionnelles qui ont présidées à la mise en place d'architectures logicielles, matérielles et opérationnelles utilisées au sein du monde automobile. Ensuite, nous détaillons leurs historiques, leurs tendances et leurs limitations et proposerons une nouvelle architecture logicielle et opérationnelle : **l'architecture virtualisée**.

1.1 Systèmes du monde automobile

Les premiers composants électroniques ont été introduits au sein des véhicules dans les années 1900, lors de l'apparition des premiers systèmes de divertissement. Longtemps limité à l'autoradio, les systèmes de divertissements ont progressivement évolué pour intégrer de nouvelles fonctionnalités (vidéo, GPS, wifi…) qui sont fournies par un système logiciel et matériel désigné sous le nom d'info-divertissement ou ***In-Vehicle-Infotainment***.

Une nouvelle classe de systèmes est apparue dans le courant des années 1990 lorsque l'industrie automobile a modifié la conception des véhicules pour y introduire des composants électroniques pilotés par des logiciels afin de remplacer ou de suppléer les composants mé-

caniques : c'est l'avènement de la **mécatronique**. La mécatronique se définit comme étant une « démarche visant l'intégration en synergie de la mécanique, l'électronique, l'automatique et l'informatique dans la conception et la fabrication d'un produit en vue d'augmenter et/ou d'optimiser sa fonctionnalité » [89].

Historiquement, une séparation fonctionnelle a été effectuée entre les systèmes mécatroniques et les systèmes d'info-divertissement. La défaillance du premier type de système peut avoir des conséquences catastrophiques sur l'environnement, notamment, en termes de pertes de vies humaines. Ils intègrent donc, dans leur processus de spécification, de conception, de développement et de test, des contraintes relevant de la sûreté de fonctionnement, du temps réel et de l'embarquabilité. Le deuxième type de système est soumis à des contraintes temps réel et de sûreté de fonctionnement moins strictes, une défaillance ayant un impact limité à l'image de marque du constructeur. En revanche, il doit répondre à de fortes contraintes d'adaptabilité pour intégrer de nouvelles et complexes fonctionnalités à un rythme soutenu. Cette division fonctionnelle s'est traduite par une différenciation dans l'architecture logicielle et matérielle utilisée pour implanter chacun des deux types de systèmes au sein des véhicules.

Dans cette section, nous allons présenter les systèmes mécatroniques et de divertissement du consommateur utilisés dans le monde automobile. Nous détaillerons, tout d'abord, l'architecture mise en œuvre dans la partie mécatronique du véhicule : l'architecture **Électrique et Électronique**. Ensuite, nous évoquerons l'architecture utilisée dans les systèmes de divertissement du consommateur. Enfin, nous présenterons les tendances prospectives qui semblent se développer quant aux évolutions du monde automobile et les limitations des architectures existantes.

1.1.1 Architecture Électrique/Électronique

Au sein de la mécatronique, l'architecture Électrique et Électronique regroupe l'ensemble des composants matériels et logiciels utilisés pour interagir avec les organes mécaniques et hydrauliques du véhicule afin de répondre aux prestations définies dans le cahier des charges.

1.1.1.1 Architecture opérationnelle fédérée

Historiquement [97], l'architecture Électrique et Électronique des véhicules a été dirigée par les capacités techniques et économiques du matériel disponible ainsi que par les contraintes organisationnelles de l'industrie automobile.

La faible puissance de calcul des premières ECUs disponibles couplée à la volonté des constructeurs automobiles de vouloir identifier clairement le responsable d'une fonctionnalité a conduit à la mise en œuvre d'une architecture opérationnelle dite **fédérée**. Une prestation client est mise en œuvre par un logiciel implanté en exclusivité sur une ou plusieurs ECUs. Le développement de chaque fonctionnalité client est confié à un unique fournisseur qui engage sa propre responsabilité et développe de manière autonome une pièce « boite noire » contenant le logiciel et le matériel requis pour implanter la prestation. Les différentes pièces sont intégrées au sein du véhicule par le réseau. En plus de la séparation claire des responsabilités, ce type d'architecture a des avantages en termes de sûreté de fonctionnement. En effet, chaque fonctionnalité

est déployée en exclusivité sur une ou plusieurs ECUs, le seul canal de propagation d'erreurs étant le réseau, ce qui facilite le confinement des erreurs et l'isolation des fautes.

En revanche, elle entraîne une multiplication des calculateurs [91, 97], au fur et à mesure que les fonctionnalités intégrées au sein des véhicules croissent, se traduisant par une augmentation de la consommation électrique et de la dissipation thermique, par la mise en œuvre d'une lourde infrastructure réseau pour faire communiquer les ECUs, le tout entraînant une augmentation du poids (Cablage, nombre d'ECUs) et des coûts. De plus, ce type d'architecture favorise une faible réutilisabilité du logiciel, qui est développé par chaque fournisseur de manière indépendante, ce qui entraîne une non mutualisation des coûts de conception, d'ingénierie et de test des logiciels.

1.1.1.2 Architecture opérationnelle intégrée

L'émergence de calculateurs plus puissants a permis le passage d'une architecture opérationnelle fédérée à une architecture opérationnelle dite **intégrée**, dans laquelle, des composants logiciels issus de fournisseurs hétérogènes et codés pour répondre à des fonctionnalités clientes différentes, sont regroupés au sein d'une même ECU. Le but est de diminuer les coûts d'acquisition et de maintenance de l'architecture matérielle (nombre d'ECUs, poids, volume et coût de câblage, consommation électrique et thermique...) qui impactent la marge opérationnelle de chaque véhicule vendu. Les industriels cherchent également à réduire les coûts de conception, d'ingénierie et de test en réutilisant les composants logiciels existants déjà testés et certifiés.

La mise en œuvre d'une architecture opérationnelle intégrée visant à incorporer des applicatifs issus de différents fournisseurs ne peut être faite sans la présence d'une standardisation commune à tous les acteurs. AUTOSAR (AUTomotive Open System ARchitecture) [13] est un consortium regroupant des constructeurs et équipementiers du monde automobile ainsi que des industriels du monde de l'électronique, des semi-conducteurs et du logiciel qui a été créé pour développer un standard commun afin de tirer profit des architectures intégrées. Son principal objectif est de faciliter la réutilisation de composants logiciels provenant de multiples fournisseurs dans différentes plates-formes véhicules en réalisant l'indépendance entre l'architecture logicielle et l'architecture matérielle. Le standard propose donc une architecture logicielle divisée en trois couches.

La couche applicative est composée de composants logiciels nommés SWC (*SoftWare Component*).

Les couches basses, appellées BSW (*BasicSoftWare*), contiennent les pilotes de l'ECU cible et du microcontrôleur, le système d'exploitation ainsi que des services logiciels et de communication.

Le RTE (*RunTime Environnment*) est un code généré pour interconnecter les appels effectués par les SWC aux services correspondant présent dans le BSW. Il gère l'abstraction de la couche matérielle vis-à-vis des composants logiciels applicatifs.

Le consortium propose également une méthode de développement pour concevoir l'architecture logicielle d'une application, décrire l'architecture matérielle et passer à une architecture opérationnelle en utilisant le standard AUTOSAR.

L'architecture intégrée abolit la séparation physique qui préexistait entre les composants logiciels résultant en un accroissement de la complexité de l'intégration et de la mise en œuvre de la sûreté de fonctionnement. La norme ISO 26262 [65] est la norme de référence pour la sûreté de fonctionnement dans le domaine automobile. Elle recommande des méthodes et mécanismes, applicables durant toutes les phases de développement du véhicule, pour atteindre et justifier la sûreté de fonctionnement. La norme préconise d'effectuer une phase d'analyse des risques pour identifier les situations dangereuses et les classifier en 4 niveaux de criticités nommés **ASIL** (*Automotive Safety Integrity Level*) allant du moins critique (ASIL A) au plus critique (ASIL D). L'attribution des niveaux de criticités prend en compte les paramètres de sévérité pour définir les conséquences de l'incident (« pas de blessés », « faiblement blessés », « blessés grave ou décès »), la fréquence d'occurrence de l'événement (« rare », « quelquefois », « assez souvent », « souvent ») et la contrôlabilité du véhicule lors de l'événement (« contrôlable », « normalement contrôlable », « incontrôlable »).

1.1.1.3 Contraintes architecturales

L'architecture Électrique/Électronique est soumise, lors de la rédaction du cahier des charges, à de nombreuses contraintes incluant notamment, des contraintes temporelles, des contraintes de coûts et d'embarquabilité. En effet, les ECUs doivent être capables de tolérer des champs magnétiques [5], de résister à de fortes variations de températures, d'assurer une dissipation thermique passive et doivent être faiblement consommateurs d'énergie. Nous allons maintenant nous focaliser sur les différentes contraintes temporelles qui sont prises en compte lors de la conception des systèmes mécatroniques. Ensuite, nous détaillerons les différents impacts que les contraintes ont sur les architectures matérielles utilisées.

Contraintes temps réel Par essence même, les systèmes mécatroniques ont été conçus pour interagir avec le véhicule en suivant un modèle de **capteurs** et d'**actionneurs**. Des capteurs sont utilisés pour récupérer des informations sur le véhicule et son environnement qui sont transmises aux calculateurs. Le logiciel présent au sein des ECUs utilise ces informations pour déduire les commandes qu'il faut appliquer aux actionneurs afin de remplir la fonction voulue. Le temps qui s'écoule entre le moment où les données sont acquises par les capteurs et où le résultat des calculs du logiciel est transmis aux actionneurs est une donnée critique. Si les commandes transmises aux actionneurs arrivent trop tardivement, le bon fonctionnement de la prestation qui doit être assurée par le calculateur n'est plus garanti. La justesse d'un calcul dépend donc non seulement de l'exactitude du résultat produit mais aussi de la durée écoulée pour produire le résultat.

Une application temps réel est usuellement composée d'un ensemble de tâches qui coopèrent entre elles. Les tâches sont activées à intervalles réguliers (**Tâches périodiques**) ou sur réception d'un événement (**Tâches apériodiques**) et peuvent comporter une **échéance** qui spécifie le temps maximal alloué à une tâche pour réaliser un traitement donné.

Pour assurer le bon fonctionnement du système, il est nécessaire d'apporter des garanties sur le respect des échéances temporelles des tâches qui sont ordonnancées. Un système est dit **prédictible** si l'on est capable de prouver qu'à l'exécution, les contraintes temporelles

(échéances, dates d'activation ...) des tâches ordonnancées sur ledit système vont être respectées. Des analyses d'ordonnançabilité sont donc effectuées sur les tâches ordonnancées au sein du système pour assurer qu'elles respectent leurs échéances. Pour être mises en œuvre, ces techniques nécessitent une connaissance précise des temps d'exécution des différents composants logiciels. Des techniques d'évaluation **de pire temps d'exécution** ou *Worst Case Execution Time* sont donc couramment utilisées pour adresser ce problème. Le *WCET* d'une tâche peut être défini comme une durée estimée au-delà de laquelle la tâche aura terminé son exécution et ce indépendamment de ses paramètres d'entrée et de la configuration initiale du matériel.

Une classification [121] des tâches en trois catégories peut être effectuée à partir des contraintes temporelles issues du monde physique. Les tâches dont les échéances doivent absolument être respectées par le système sont qualifiées de **temps réel dur**. On parle de **temps réel ferme** lorsque le dépassement de l'échéance d'une tâche ne provoque pas de conséquences catastrophiques sur le système, les résultats produits par la tâche après l'échéance sont alors inutiles. Une tâche est dite **temps réel mou** lorsque l'utilité des calculs qu'elle produit décroît au cours du temps une fois que l'échéance a expiré.

Implication sur les architectures matérielles La mise en œuvre d'un système prédictible nécessite le calcul de bornes temporelles exactes portant sur le temps d'exécution des tâches. L'obtention de ce résultat est lié à l'architecture matérielle sur laquelle les tâches sont exécutées mais dépend également des données d'entrées du système et des interactions du système avec l'environnement. De manière générale, l'espace d'états des données d'entrées et l'espace d'états du matériel sont, le plus souvent, trop larges pour être explorés exhaustivement. Il n'est alors pas possible de déterminer, de manière exacte, le pire temps d'exécution (*WCET*) des tâches. L'analyse temporelle du système est donc effectuée en mettant en œuvre des abstractions simplifiées et conservatives de la plate-forme d'exécution. Ces abstractions, qui provoquent une perte d'informations, sont responsables des différences entre les *WCETs* estimés et les *WCETs* mesurés.

Les architectures matérielles modernes utilisées dans les microprocesseurs mettent en œuvre des techniques de caches, de pipelines, d'exécution spéculative qui contribuent largement aux performances moyennes desdites architectures. La conséquence annexe de ces techniques résulte en une haute variabilité dans les temps d'exécution des instructions et en une complexification des techniques de calcul de *WCET*, ce qui se traduit par un fort écart entre le pire temps mesuré lors de l'exécution d'un programme et le pire temps estimé. In-fine on assiste à un surdimensionnement de la plate-forme matérielle.

Les architectures temps réel matérielles utilisées par l'industrie automobile cherchent à minimiser la variabilité pour garantir une bonne prédictibilité tout en garantissant de bonnes performances.

1.1.2 Architecture des systèmes de divertissement du consommateur

Les systèmes de divertissement du consommateur ont été les premiers, dans le milieu des années 1920, à introduire des composants électroniques au sein des véhicules sous la forme

de récepteurs radio à lampes. Si l'apparition de transistors, dans les années 1950, a contribué à l'amélioration des capacités techniques des appareils et à la diffusion massive des autoradios au sein des automobiles, le concept de base a peu évolué jusqu'à la fin des années 1970. L'introduction des premiers systèmes de navigation dans les années 1980, puis des systèmes multimédia dans les années 2000 a changé la donne. Désormais, la façade de l'autoradio est nommée *head unit* et concentre 70% du code du véhicule.

Le système multimédia moderne a un rôle qui va au-delà de celui du simple autoradio : il sert d'interface entre le véhicule et le consommateur, contribuant ainsi à l'expérience utilisateur et devenant un des principaux critères de choix lors de l'achat du véhicule. Il est devenu un domaine que les constructeurs ne peuvent ni ne veulent abandonner.

A contrario des systèmes mécatroniques, où chaque constructeur dispose d'un quasi-monopole sur les solutions qu'il décide d'intégrer dans son véhicule, les systèmes de divertissement du consommateur ont, de tout temps, été soumis à la concurrence agressive des produits issus du marché de l'après-vente (« After market ») et maintenant de l'industrie du marché grand public (« Consumer markets »).

1.1.2.1 De l'autoradio aux systèmes multimédia

Des années 1920 aux années 2000, l'autoradio est le principal système de divertissement des conducteurs et passagers présent au sein des véhicules. Implantés au départ en utilisant un format propre à chaque constructeur, la mise en œuvre de la norme DIN 75490 [66] en 1984 standardise le montage des autoradios et contribue à la création d'un marché d'après-vente qui concurrence les solutions proposées par les constructeurs et équipementiers.

Les premiers systèmes de navigation apparaissent dans les années 1980-90 (GPS CARMINAT sur Safrane en 1995). Réservés aux véhicules haut de gamme, ils utilisent des écrans CRT couleurs pour afficher les trajets, les cartes de navigation sont stockées sur CDROM et la position du véhicule est calculée en utilisant des technologies de reconnaissance cartographique (« map matching ») nécessitant un accès à des capteurs du véhicule (vitesse, accéléromètre, capteur angulaire au niveau de la direction, ...) couplé aux premiers récepteurs de GPS [64]. Les volumes de ventes sont faibles et les constructeurs ont du mal à rentabiliser leurs coûts de développement et de fabrication. En mai 2000 [51], le président Clinton prend la décision d'arrêter la dégradation volontaire du service civil du GPS, qui était effectuée pour des raisons de sécurité nationale, augmentant ainsi la précision du GPS de 100 à 10 mètres. Les systèmes de navigation, qui n'ont plus besoin des techniques de reconnaissance cartographique intégrées au véhicule pour être opérationnels, se diffusent alors massivement aux utilisateurs du monde automobile sous la forme de *Personal Navigation Devices* issus des marchés de l'après-vente. Ils introduisent une nouvelle ergonomie associant écran plat couleur et interface tactile.

A la fin des années 2000, la diffusion massive des smartphones modifie les attentes des consommateurs qui souhaitent désormais pouvoir accéder aux mêmes services dans leurs voitures que dans leurs téléphones. Les constructeurs et équipementiers sont donc mis en concurrence avec des acteurs du monde du marché grand public qui bénéficient de gigantesques volumes de production générant d'importants capitaux et qui sortent de nouveaux produits à

un rythme soutenu, stimulant ainsi l'appétit insatiable des consommateurs pour de nouvelles fonctionnalités. Le système multimédia, nommé aussi IVI (*in-vehicle infotainment*) devient un important différenciateur, pour les clients, lors de l'acte d'achat du véhicule. Une des raisons du succès des acteurs du monde du marché grand public réside dans la richesse de leur écosystème applicatif. Par le déploiement de marchés d'applications (Apple store, Google Play, ...) couplé à la diffusion de kits de développement, les éditeurs de plates-formes du marché grand public ont mis en œuvre une politique incitative pour encourager les développeurs à créer de nouvelles applications. Le nombre final d'applications ainsi développées dépasse largement celui qui aurait pu être financé par les fonds propres des éditeurs.

Les systèmes multimédia des véhicules modernes doivent également composer les contraintes de développement du monde de l'informatique grand public avec les contraintes du monde automobile. Les nouvelles fonctionnalités introduites au sein des véhicules ne doivent pas mettre en danger le conducteur que ce soit par un manque d'ergonomie, de sûreté ou de sécurité. Le matériel utilisé dans les voitures est soumis à des contraintes physiques (Températures, Électromagnétisme, vibrations) qui sont largement plus destructrices que celles subies par les téléphones. Il doit donc être conçu et certifié [5] pour pouvoir tolérer ces contraintes ce qui entraîne une augmentation des coûts et, de fait, un allongement de sa durée d'utilisation pour le rentabiliser. Enfin, alors que le rythme de renouvellement des smartphones et donc des applications associées est en moyenne de 2 ans [152], les constructeurs doivent garantir la pérennité de leurs produits pendant toute la durée de vie du véhicule (10 ans après le dernier véhicule sorti de chaîne soit au total une durée moyenne de 15 ans).

Face aux contraintes des systèmes multimédia, les acteurs du monde automobile sont en difficulté. En effet, les solutions historiques de divertissement de l'automobile sont basées sur des systèmes propriétaires développés en propre par chaque équipementier. La prise en compte des demandes des consommateurs entraîne donc une augmentation exponentielle de la quantité de logiciel à produire et des coûts de développement, de validation et de test associés. Les équipementiers se retrouvent à la traîne et essayent dans une perpétuelle course de rattraper les innovations technologiques générées par l'industrie du marché grand public. Ce problème est exacerbé par le mode de fonctionnement interne à l'industrie automobile où chaque équipementier développe sa propre solution, et concourt contre les autres fournisseurs afin d'être sélectionné par les constructeurs automobiles pour, au final, vendre un faible nombre de périphériques. Aucun acteur du monde automobile ne dispose d'assez d'influence pour devenir attractif et inciter les développeurs d'applications à développer pour leur solution propriétaire ce qui entraînerait une réduction de leurs coûts de développement.

Pour faire face à ces difficultés, les constructeurs et équipementiers ont créé un consortium nommé GENIVI Alliance qui regroupe 180 industriels afin de définir un standard logiciel, réutilisant des solutions libres existantes, applicable au multimédia du monde automobile.

1.1.2.2 Contraintes architecturales

L'évolution des fonctionnalités intégrées au sein des systèmes multimédia s'est traduite par une augmentation en besoin de puissance de calcul, les systèmes IVI actuels des véhicules devenant plus puissant que la plupart des calculateurs mécatroniques du véhicule. Pour faire face

à cette demande, les architectures matérielles désormais embarquées au sein des calculateurs multimédia utilisent majoritairement des processeurs multi-cœurs et mettent en œuvre des techniques matérielles de maximalisation des performances (Unité de préchargement, Unité de prédiction de branchement, caches, ...) qui se font au détriment de la prédictibilité du matériel.

Le système multimédia est également assujetti à des contraintes d'embarquabilité : taille, dissipation passive des composants, résistance aux vibrations et aux perturbations électromagnétiques, et à de forts impératifs de coûts. Les fabricants matériels ont donc opté pour une approche **COTS** (*Commercial Off-The-Shelf*) consistant à réutiliser des architectures matérielles embarquées déjà développées pour des applications multimédia et à les décliner pour le monde automobile (Ajout de périphériques, certification ECM, ...). A ce titre, nous avons utilisé au sein de ma thèse, la carte embarquée i.MX 6 SABRE Lite [119] une plate-forme COTS conçue pour exécuter des systèmes multimédia.

Les systèmes multimédia sont également sujets à des contraintes temps réel molles principalement liées aux temps de réponse nécessaires pour assurer l'affichage des contenus multimédia et la réactivité nécessaire aux interactions avec les utilisateurs. Au vu de la faiblesse des exigences ces contraintes ont des impacts minimaux sur la conception des architectures matérielles.

1.2 Tendances, limitations et solutions

Les besoins et les fonctionnalités du monde de l'automobile et du monde de l'embarqué sont en constante évolution. Or, les normes et les standards sont développés pour répondre aux évolutions passées et non aux tendances à venir. Elles font des choix techniques qui, logiques à un instant donné, peuvent ne plus être en adéquation avec les besoins futurs.

Dans cette section, nous évoquerons les évolutions prévisionnelles des besoins et fonctionnalités du monde automobile. Ensuite, nous détaillerons les limitations des architectures et standards actuellement utilisés. Enfin, nous proposerons des solutions pour combler les lacunes précédemment mises en évidence.

1.2.1 Tendances

Deux tendances se dégagent quant à l'évolution des systèmes embarqués dans les véhicules. Tout d'abord, nous observons une augmentation croissante du nombre de fonctionnalités contrôlées par le logiciel tant au sein des systèmes mécatroniques que des systèmes d'info divertissement. De plus en plus de fonctionnalités antérieurement mises en œuvre par de la mécanique ou par l'humain sont désormais suppléées ou remplacées par des logiciels. Nous constatons également que la séparation, originellement présente entre les systèmes mécatroniques et les systèmes multimédia, tend à s'estomper. Désormais, les interfaces tactiles des systèmes multimédia se sont substituées aux interfaces mécaniques (Boutons, manettes, ...) contrôlées par les systèmes mécatroniques.

1.2.1.1 Une augmentation des fonctionnalités logicielles

Les systèmes mécatroniques introduits dans les années 1990 avaient pour objectif de suppléer aux organes mécaniques du véhicule en fournissant une assistance électrique contrôlée par du logiciel et ce, pour augmenter les performances et les fonctionnalités des produits. Le modèle économique actuel laisse à penser que cette dynamique va se poursuivre. Les véhicules haut de gamme produits par les constructeurs de luxe doivent sans cesse intégrer de nouvelles prestations supplémentaires pour justifier leur niveau de prix et se distinguer de la concurrence. Cette évolution du niveau de services se répercute in fine sur les constructeurs généralistes qui doivent, à terme, intégrer les mêmes services avec comme exigence supplémentaire un prix contenu.

Le modèle normatif actuel pousse également au développement des fonctionnalités logicielles. Le respect des normes européennes d'émission, dites normes **EURO** [40] passe par des développements logiciels de plus en plus poussés.

L'apparition des systèmes d'aide à la conduite (*ADAS : Advanced Driver Assistance Systems*) représente une nouvelle étape quant à l'augmentation des prestations ajoutées dans l'architecture fonctionnelle. Ces systèmes qui ont été introduits dans le milieu des années 1990 [20, 45] utilisent des capteurs de l'environnement (radar, caméra, …) afin d'assister le conducteur aussi bien pour l'alerter de l'apparition d'une situation accidentogène que pour le libérer de certaines tâches de conduite manuelle du véhicule. L'utilisation d'ADAS pour suppléer ou remplacer les décisions et les actions humaines permet l'élimination des erreurs du conducteur, qui sont à l'origine de 93% [39] des accidents, engendrant une baisse des accidents de la route et un meilleur contrôle du véhicule. Les ADAS sont donc un enjeu de sécurité majeur et vont devoir être déployés sur l'ensemble des gammes de véhicules à un coût maîtrisé. En 2014, le département de la surveillance de la sécurité des transports des États Unis (NHTSA) a rendu obligatoire la présence d'une caméra de recul sur tous les véhicules de moins de 4,5t d'ici mai 2018 [96]. Le véhicule totalement autonome représente l'aboutissement des systèmes ADAS actuels. Les feuilles de route des constructeurs (Audi, BMW, Daimler, Ford, GM, Google, Kia, Mercedes-Benz, Renault-Nissan, Tesla, Toyota) prédisent l'arrivée des premiers véhicules autonomes, tout au moins sur une partie de leur trajet, en 2020 [18, 31, 38, 42, 44, 71, 131].

Les systèmes multimédia existants assurent un divertissement global à l'échelle de l'ensemble des passagers du véhicule. A l'heure actuelle, où une part croissante de la population française est équipée d'un smartphone, la présence d'un seul système multimédia pour tous les passagers du véhicule n'est plus suffisante. Chaque passager va devoir accéder à son programme dédié d'info-divertissement et va vouloir contrôler la partie du véhicule avec laquelle il interagit physiquement.

1.2.1.2 Des systèmes de plus en plus intégrés

Si la séparation fonctionnelle et architecturale, initialement mise en place entre les systèmes mécatroniques et les systèmes de divertissement, était conceptuellement légitime et concrètement effective, aucun canal de communication n'existant entre les systèmes, la frontière entre les deux types de systèmes s'est peu à peu estompée. Les premiers systèmes de localisation

par GPS avaient besoin d'informations issues des organes mécaniques du véhicule (Capteur d'angle de volant, de vitesse de roue, ...). Dans les nouveaux véhicules, des fonctionnalités, traditionnellement dévolues aux systèmes mécatroniques, sont désormais gérées par les systèmes multimédia qui contrôlent une partie des organes du véhicule (climatisation, sièges...).

L'intégration fonctionnelle des deux types de systèmes va sans doute s'accélérer. En effet, comparativement aux interfaces des systèmes IVI, les interfaces homme-machine présentes dans les systèmes mécatroniques actuels sont relativement basiques. Elles se fondent sur une combinaison bien établie d'afficheurs (tableau de bord), de boutons et de manettes pour interagir avec le véhicule. L'influence des systèmes multimédia, qui fait usage d'interfaces tactiles personnalisables couplées à des projections en trois dimensions, se répercute sur les interfaces homme-machine des systèmes mécatroniques qui vont devoir offrir les mêmes fonctionnalités. L'intégration fonctionnelle est également un moyen pour les constructeurs de se différencier des systèmes d'info-divertissement proposés par les industriels de l'électronique grand public. Dans le futur, les deux classes de systèmes vont donc partager des périphériques (Écrans, affichage tête haute, antennes intelligentes, systèmes sonores, caméra, ...), des composants matériels (GPU, ...) et des fonctionnalités logicielles (Interface homme-machine, ...).

On observe également des tendances d'intégration au niveau de la connectivité. Les voitures modernes disposent toutes de moyens de communication filaires (prise OBD) et, pour certaines, de connections sans fil (2G, 3G, 4G, Wi-Fi, Bluetooth). Actuellement, ces connectiques sont séparées, les connectiques filaires étant utilisées par les systèmes mécatroniques pour le débogage et les mises à jour, et les connectiques sans fil étant utilisées par les systèmes multimédia. L'arrivée de nouvelles normes de communications inter-véhicules [24] (« Vehicle to Vehicle ») ou entre véhicules et équipements d'infrastructures (« Vehicle to Infrastructure »), basées sur des technologies sans fils, à but de coopération va mettre fin à cette séparation. Les deux types de systèmes vont devoir accéder aux mêmes périphériques matériels.

1.2.2 Limitations

Les tendances vers lesquelles s'achemine le monde automobile mettent en évidence des limitations dans les solutions et les standards actuellement utilisés. Ces limites peuvent être classées en trois catégories : les limitations de l'architecture matérielle, les limitations de l'architecture logicielle et les limitations de l'architecture opérationnelle.

1.2.2.1 Limitations de l'architecture matérielle

La multiplication des fonctionnalités présentes dans les véhicules a de forts impacts sur l'architecture matérielle, se traduisant par une demande accrue en puissance de calcul et par une congestion des réseaux.

Le développement des véhicules autonomes en est le parfait exemple. A contrario des systèmes mécatroniques classiques qui utilisent des capteurs basiques, les véhicules autonomes vont devoir intégrer de nouveaux capteurs complexes (LIDAR [1], Ultrason, Caméra, ...) pour

1. *Light Detection and Ranging* : technique de mesure à distance fondée sur l'utilisation d'un faisceau de lumière renvoyé vers son émetteur.

récupérer des informations sur l'environnement. Une importante capacité de calcul est donc nécessaire pour analyser et fusionner les données recueillies par les différents capteurs et effectuer la prise de décision. Actuellement, les prototypes de véhicules autonomes utilisent des architectures matérielles de type station de travail avec des processeurs quatre-cœurs [39] qui sont incomparablement plus puissants que la majeure partie des processeurs embarqués dans les calculateurs véhicules.

Cette demande en puissance de calcul est exacerbée par les exigences en termes de sûreté de fonctionnement qui sont associées à une partie des services. En effet, les mécanismes permettant de garantir le bon fonctionnement des services (les mécanismes de contrôle de vraisemblance, la réplication des fonctionnalités, ...) sont à l'origine d'une augmentation des besoins en ressources CPU et mémoire. Les techniques d'analyse d'ordonnançabilité et de calcul de pire temps d'exécution entraînent un surdimensionnement de la plate-forme matérielle se traduisant également par une augmentation théorique de la puissance de calcul nécessaire.

1.2.2.2 Limitations de l'architecture logicielle

Les standards logiciels utilisés dans le monde de l'automobile disposent de points forts et de points faibles. AUTOSAR a été conçu pour développer des applications temps réel embarquées dans des calculateurs disposant de faible taille mémoire, de faibles ressources CPU et communiquant avec des capteurs et actionneurs rudimentaires. Le système est configuré de façon statique : toutes les structures de données utilisées sont connues, déclarées à l'avance et initialisées statiquement. Le code des différentes couches (Applicatif, RTE, OS) est compilé en un seul binaire, déployé en ROM ou en Flash, mais n'a pas besoin d'être stocké en RAM. Si cette configuration favorise l'embarquabilité du logiciel, elle complique les mises à jour des applicatifs et de l'OS, toute mise à jour d'un élément de la configuration entraînant la mise à jour et le redéploiement du binaire sur le calculateur.

AUTOSAR n'a pas été conçu pour échanger des informations avec les périphériques relativement complexes que sont les caméras, radars, GPU et écrans. Il n'offre donc aucune facilité quant à la réalisation d'interfaces homme-machine utilisant l'accélération matérielle. Les interfaces homme-machine réalisées sous AUTOSAR sont donc développées sous la forme d'un composant logiciel nommé « Complex Device Driver », transverse aux couches définies dans le standard. Ce type de composant a été conçu pour encapsuler des modules logiciels dont l'intégration, dans le standard, n'a pas été normalisée. La création de ce type de composant est complexe, la pile graphique nécessaire à la mise en œuvre des fonctionnalités d'accélération matérielles en 3D étant particulièrement compliquée.

De la même manière, les systèmes actuels d'ADAS ne sont pas développés en utilisant une architecture intégrée, tel que proposée par AUTOSAR, mais en utilisant une architecture fédérée. Les capteurs d'environnement, dits « intelligents » [25] qui perçoivent et analysent l'environnement, sont développés en utilisant leurs propres standards logiciels et matériels. Ils communiquent les informations obtenues au système de prise de décision qui est conforme au standard AUTOSAR. Si cette solution est fonctionnelle avec les systèmes d'ADAS actuels, elle risque de ne plus l'être avec le véhicule autonome où il va falloir traiter et fusionner des données issues de multiples capteurs. Les prototypes des véhicules autonomes utilisent Linux

comme système d'exploitation [39] avec un modèle de tâches à criticité multiples qui comprend un petit nombre de tâches temps réel ordonnancées en parallèle de tâches *best effort*. Si l'utilisation de Linux comme système d'exploitation dans les véhicules autonomes, règle un certain nombre de problèmes, elle pose des problèmes de sûreté de fonctionnement.

Le standard GENIVI, basé sur Linux, a été conçu pour exécuter des applications multimédia. Il est donc parfaitement adapté pour gérer les interfaces homme-machine et utiliser le GPU. Conçu avec une architecture logicielle dynamique, facilitant ainsi les mises à jour à distance, il a été utilisé comme démonstrateur pour implanter des tableaux de bord véhicule. En revanche, la quantité de code utilisée dans le système est considérable et ne respecte pas les règles de conception et de développement [65, 129] couramment utilisées dans les systèmes critiques. La mise en place de politiques de sûreté de fonctionnement sur des systèmes tels que le tableau de bord, classifié au niveau de criticité ASIL B, est donc un défi.

1.2.2.3 Limitations de l'architecture opérationnelle

Si la séparation fonctionnelle, établie entre les systèmes mécatroniques et les systèmes de divertissement n'est plus de mise, la différenciation dans l'architecture logicielle et matérielle est, quant à elle, restée. L'intégration entre les deux types de systèmes, développés avec des standards logiciels et des architectures de calculateurs différents, est actuellement réalisée en utilisant une architecture fédérée. Chaque système est déployé sur des ECUs dédiées qui sont reliées par une passerelle, nommée *Telecommunication Unit*, assurant le filtrage des communications inter-calculateurs. Si ce type d'architecture a des avantages en termes de sécurité et de sûreté de fonctionnement, elle limite le partage de ressources matérielles, tels que les écrans et le GPU, qui sont nécessaires à une intégration plus poussée. En effet, toute ressource partagée sur de telles architectures est allouée en exclusivité à un des calculateurs, le partage étant effectué par le réseau, ce qui pose des problèmes de latence et de débit.

1.2.3 Solutions : virtualisation sur des systèmes multi-cœurs

Les architectures matérielles actuelles utilisées par les systèmes mécatroniques ne suffisent plus à atteindre le niveau de performances nécessaires à la réalisation des fonctionnalités qui seront implantées dans les prochains véhicules. L'utilisation d'architectures matérielles de type COTS, actuellement utilisées dans le calculateur le plus puissant du véhicule : le calculateur multimédia, permet de répondre à cette problématique tout en gardant un coût maîtrisé. Les futures architectures embarquées COTS de type « many core » qui regroupent jusqu'à 256 cœurs [92] apparaissent comme des alternatives crédibles pour remplacer ou suppléer les futures architectures matérielles du monde automobile.

La virtualisation apparaît être une solution viable pour répondre à certain des problèmes rencontrés au sein de l'architecture logicielle, matérielle et opérationnelle actuellement utilisée. Il s'agit de faire cohabiter, sur un même calculateur, des applications et des systèmes d'exploitation encapsulé au sein de **machines virtuelles** en utilisant une couche logicielle nommée **hyperviseur** ou **moniteur de machine virtuelle** qui gère l'accès au matériel. Chaque applicatif ainsi hébergé dans une machine virtuelle, dispose de son propre ensemble de ressources

dédiées et est isolé des autres applications.

Du point de vue de l'architecture matérielle, la virtualisation offre les avantages d'une architecture intégrée poussée à l'extrême. En regroupant les fonctionnalités sur un même calculateur, elle permet de diminuer la congestion du réseau, les messages inter-machines virtuelles étant échangés en utilisant des techniques de mémoire partagées, de diminuer la consommation électrique et la dissipation thermique. L'utilisation d'hyperviseurs sur des architectures matérielle de type « many core » à de nombreux avantages. Tout d'abord, la virtualisation facilite le déploiement de systèmes d'exploitation patrimoniaux, incapable d'exploiter un grand nombre de cœurs, en partitionnant le processeur en de multiples sous domaines chacun d'entre eux contenant un nombre réduit de cœurs. L'hyperviseur peut également être utilisé pour ajouter dynamiquement des cœurs à un sous domaine qui a besoin de puissance de calcul supplémentaire ou peut gérer la consommation énergétique en supprimant des processeurs de certains domaines et en éteignant des cœurs IDLE.

La virtualisation dispose de nombreux avantages en termes d'intégration logicielle. Elle favorise la cohabitation, sur une même plate-forme, d'applicatifs issus du monde de la mécatronique basés sur le standard AUTOSAR, d'applicatifs issus du monde du multimédia, basé sur le standard GENIVI, et d'applicatifs développés en utilisant de nouveaux standards. Elle permet, de ce fait, de profiter des forces et des faiblesses de chacun des écosystèmes et facilite la réutilisation du code et l'intégration logicielle entre les standards existants. La virtualisation permet également d'exécuter plusieurs instances d'un même système d'exploitation, répondant ainsi à la problématique de personnalisation du système d'info divertissement ou chaque passager du véhicule accède à son propre système multimédia, et ce, à moindre coût. Il n'est plus en effet nécessaire d'avoir une carte par utilisateur connecté. Les systèmes actuels embarqués étant de plus en plus connectés au monde extérieur, la probabilité qu'une application ou qu'un système d'exploitation soit attaqué et compromis augmente dramatiquement. En instaurant une séparation entre les services pouvant être accédés à distance et les autres systèmes, en utilisant des machines virtuelles de filtrage, l'architecture virtualisée peut être utilisée pour limiter les conséquences d'une attaque réussie sur le système.

En terme d'architecture opérationnelle le regroupement de multiples systèmes d'exploitation et d'applications sur une même carte facilite le partage de ressources matérielles telle que le GPU, les écrans, la connectique.

1.3 Conclusion

Nous avons, dans ce chapitre, posé le contexte industriel dans lequel se déroulent nos travaux, pour ensuite énumérer les différentes contraintes auxquelles le monde automobile est confronté. Nous avons ensuite détaillé les différentes architectures logicielles, matérielles et opérationnelles utilisées par le monde automobile ainsi que leurs limites. Nous proposons, pour y remédier, une nouvelle architecture opérationnelle : l'architecture virtualisée sur processeurs multi-cœurs.

Avant de voir dans le chapitre 3 l'état de l'art des solutions qui ont été proposées, nous allons, dans le chapitre suivant, détailler le contexte technique nécessaire à la réalisation de

nos travaux et voir les différentes problématiques soulevées par la mise en œuvre d'une telle architecture.

Chapitre 2

Introduction au problème de contention mémoire

Sommaire

2.1 Virtualisation 22
2.1.1 Définitions et concepts 22
2.1.2 Virtualisation des composants 23
2.1.3 Virtualisation des systèmes embarqués 29
2.2 Multi-cœurs 30
2.2.1 Modèle de programmation multi-cœurs sur architectures virtualisées 30
2.2.2 Multi-cœurs et contraintes temps réel 31
2.3 Hiérarchie mémoire 33
2.3.1 Caractéristiques des mémoires 34
2.3.2 Caches 35
2.3.3 Mémoire principale 39

Nous avons vu, dans le chapitre précédent, qu'une architecture virtualisée était la solution envisagée par de nombreux constructeurs du monde automobile pour pallier les limitations des architectures opérationnelles existantes.

Dans ce chapitre, nous allons introduire les mécanismes nécessaires à la bonne compréhension de ce manuscrit. Nous étudierons plus particulièrement les mécanismes utilisés dans les architectures multiprocesseurs pour partager la hiérarchie mémoire entre les multiples cœurs. Nous verrons que les impacts d'un tel partage peuvent entraîner une violation des échéances lorsque des tâches temps réel sont exécutées en parallèle d'applications *best effort*.

Dans un premier temps, nous allons définir le concept même de virtualisation pour ensuite étudier les défis et les solutions que posent la mise en œuvre d'une telle technique sur les composants électroniques de la plate-forme matérielle.

Ensuite, dans une deuxième partie, nous aborderons les impacts des architectures multi-cœurs sur les contraintes temps réel intrinsèques aux systèmes mécatroniques. Nous présen-

terons les différents modèles de programmation qui ont été proposés pour exploiter de telles architectures, puis nous détaillerons les problèmes d'interférences sur le système mémoire partagé des architectures multi-cœurs qui introduisent des latences additionnelles dans les temps d'exécution des programmes temps réels.

Enfin, dans une dernière partie, nous étudierons plus en profondeur le fonctionnement des composants du système mémoire pour mettre en évidence les problèmes d'interférences temporelles qui existent.

2.1 Virtualisation

La virtualisation est une technique de partage du matériel qui permet d'ordonnancer plusieurs systèmes d'exploitation sur une même machine, les systèmes ainsi hébergés ayant un comportement quasi-identique à celui qu'ils ont lorsqu'ils sont exécutés isolément sur le matériel.

Les prémisses de la virtualisation remontent à la fin des années 60 avec notamment l'ordinateur central « IBM System/370 » le premier matériel conçu pour la virtualisation [1]. Dans les années 1990, le projet Disco [22], initié à Stanford, introduit une nouvelle technique nommée **paravirtualisation**, pour exécuter plusieurs systèmes d'exploitation sur du matériel qui n'a pas été conçu pour permettre la virtualisation. A la fin des années 1990, l'équipe de Stanford fonde la société VMware [23] qui fournit le premier produit de virtualisation pour l'architecture matérielle x86 qui n'est pas nativement virtualisable [88]. La virtualisation est alors rapidement adoptée par les industriels, spécialement dans les **centres de données** et sur les **ordinateurs de bureau**. Dans le milieu des années 2000, des fondeurs ajoutent le support de la virtualisation dans l'architecture matérielle x86 [95] diminuant ainsi la complexité des hyperviseurs et augmentant les performances. Enfin, au début des années 2000, la technologie de virtualisation a gagné le domaine de l'embarqué aussi bien dans le domaine des téléphones mobiles [17, 62] que dans des domaines plus génériques [58, 59, 102]. En 2010, l'industriel ARM annonce un support matériel de la virtualisation dans son futur processeur Cortex-A15 [136].

2.1.1 Définitions et concepts

La virtualisation est une technologie utilisée pour permettre la cohabitation de plusieurs systèmes d'exploitation sur une même machine physique en utilisant une couche logicielle, nommée **moniteur de machines virtuelles** (*Virtual Machine Monitor*) ou **hyperviseur**, qui s'intercale entre les systèmes d'exploitation invités et le matériel. On appelle **machine virtuelle** l'environnement créé par le moniteur de machines virtuelles pour héberger les systèmes d'exploitation.

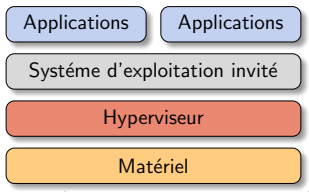
En 1974 Popek et Goldberg's [105] caractérisent le comportement d'un moniteur de machines virtuelles à l'aide de trois propriétés. La propriété d'**équivalence** implique qu'un programme exécuté dans un environnent virtuel doit avoir un comportement identique à celui obtenu lorsqu'il est exécuté nativement, à deux exceptions près. Tout d'abord, l'exécution occasionnelle du moniteur de machines virtuelles peut provoquer des différences dans les temps

d'exécutions des programmes virtualisés. Ensuite, le partage des ressources matérielles, entre les programmes virtualisés, peut entraîner des différences dans le comportement desdits programmes qui peuvent essayer d'accéder à la même quantité de ressources que lorsqu'ils sont exécutés de manière non virtualisée. La deuxième préconisation spécifie qu'un **sous-ensemble statistiquement dominant des instructions des programmes doit être exécuté directement par le processeur** garantissant ainsi de bonnes performances. Enfin, le moniteur doit avoir un **contrôle total** sur les ressources matérielles du système.

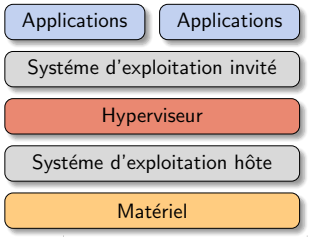
En 1973, golderberg [49] effectue une classification des moniteurs de machines virtuelles en deux catégories :

Type 1 Les hyperviseurs de type 1 (Figure 2.1a) appelés également **natifs** ou *bare metal* qui s'exécutent directement sur le matériel (Xen [16], PikeOS [104], OKL4 [98], INTEGRITY Multivisor [63]).

Type 2 Les hyperviseurs de type 2 (Figure 2.1b) s'exécutent sous la forme d'applications ordonnancées par un système d'exploitation dit **système hôte** (VirtualBox [142], VMware Workstation [23]).



(a) Hyperviseur de type 1 (natif)



(b) Hyperviseur de type 2

FIGURE 2.1 – Types d'hyperviseurs

2.1.2 Virtualisation des composants

Popek et Goldberg [105] décrivent les caractéristiques qui doivent, à minima, être implantées dans une architecture matérielle pour pouvoir exécuter des machines virtuelles en respectant les propriétés énoncées ci-dessus.

Tout d'abord, le processeur doit être doté d'au moins deux modes d'exécution, un **mode privilégié** et un **mode utilisateur**. Une distinction est faite entre les instructions **privilégiées** qui, comme leur nom l'indique, peuvent être exécutées uniquement lorsque le processeur est en mode privilégié, et les instructions **non privilégiées** qui peuvent être exécutées quel que soit le mode du processeur. Toute tentative d'exécuter le premier type d'instruction alors que le processeur est en mode non privilégié se traduit par le déclenchement d'une **exception** (*trap*) qui interrompt le processeur et empêche l'instruction litigieuse de s'exécuter.

Ensuite, la machine doit être pourvue d'un mécanisme de **mémoire virtuelle**. Les logiciels exécutés sur la machine perçoivent l'espace mémoire comme un unique espace d'adressage contigu, nommé **espace d'adressage virtuel** qui est découplé de **l'espace d'adressage**

physique. Le système d'exploitation configure des structures de données pour établir la correspondance entre l'espace d'adressage virtuel et l'espace d'adressage physique. Le processeur émet des **adresses virtuelles** qui sont traduites par l'unité de gestion de la mémoire (*Memory Managemnt Unit*) vers des **adresses physiques** en utilisant la structure de correspondance configurée par le système d'exploitation. Le système de mémoire virtuelle doit également être couplé à un système de **protection mémoire** apte à déclencher une **exception** lorsqu'un accès à une zone mémoire illégale est effectué.

Enfin, lorsqu'une exception matérielle est déclenchée, le processeur doit pouvoir donner la main à une routine d'exécution et, une fois la routine terminée, doit être capable de totalement restaurer le contexte d'exécution préexistant à l'exception. Dans le chapitre 5, nous utiliserons un mécanisme d'interruption dérivé du mécanisme d'exception pour contrôler l'exécution des logiciels virtualisés.

Nous allons maintenant détailler les différents composants matériels qui doivent être virtualisés.

2.1.2.1 Processeur

Les concepteurs d'hyperviseurs effectuent une classification des instructions en trois catégories. Les instructions dites **sensibles à la configuration** (*control sensitive*) incluent toutes les instructions capables de changer la configuration des ressources du système (protection mémoire, mode du processeur). Les instructions **sensibles au comportement** (*behavior sensitive*) incluent toutes les instructions dont le comportement ou le résultat varie selon la configuration des ressources du système (mode du processeur, ...). Enfin, les instructions **non sensibles** regroupent les instructions restantes. Le premier théorème de Popek et Goldberg [105] énonce que sur toute architecture matérielle qui respecte les caractéristiques définies précédemment, une machine virtuelle efficace peut être construite si toutes les instructions sensibles à la configuration et au comportement sont un sous-ensemble des instructions privilégiées.

Les systèmes d'exploitation modernes utilisent les modes d'exécutions du processeur pour effectuer une séparation, entre le code du noyau exécuté en mode privilégié qui a le droit de configurer les ressources matérielles de la machine via les instructions sensibles, et le code des applications exécuté en mode utilisateur. Dans une architecture virtualisée, où l'hyperviseur doit avoir un contrôle total sur les ressources matérielles du système, les systèmes d'exploitation invités ne peuvent être exécutés en mode privilégié. La virtualisation fait donc appel à la technique dite de *trap and emulate*. Les systèmes d'exploitation invités sont exécutés en mode utilisateur. Lorsque des instructions sensibles sont exécutées par le système virtualisé une exception est générée par le matériel. L'hyperviseur capture alors l'exception, détermine quelle est l'instruction source de l'interruption, émule son comportement éventuellemnt en effectuant un appel système dans le cas d'un hyperviseur de type 2, et redonne la main à la machine virtuelle.

L'exécution d'un logiciel dans un mode différent de celui pour lequel il a été conçu peut poser différents problèmes. Tout d'abord, certaines architectures matérielles [133] mettent à disposition des instructions qui peuvent être utilisées par l'OS invité pour récupérer le vrai

niveau de privilège sous lequel il est exécuté. Ensuite, l'utilisation des modes du processeur pour détecter les instructions sensibles peut poser des problèmes de performances. En effet, le coût temporel d'une instruction trappée et émulée étant largement supérieure à celui d'une instruction nativement exécutée, un système invité qui exécute, à une forte fréquence, des instructions privilégiées (Masquage/démasquage des interruptions, ...) subit une importante dégradation des performances. Enfin, une partie des architectures modernes actuelles (ARM [136], x86 [133]) ne respectent pas nativement le premier théorème de Popek et Goldberg et ne sont donc pas directement virtualisables. Par exemple, l'instruction x86 `popf`, utilisée pour masquer les interruptions matérielles, ne provoque pas d'exception lorsqu'elle est exécutée en mode utilisateur.

Plusieurs méthodes ont été développées pour pallier à ces inconvénients :

La technique de **traduction du code binaire** vise à remplacer, à l'exécution, dans le binaire du code noyau les instructions sensibles non privilégiées par des appels à l'hyperviseur.

La technologie de **paravirtualisation** consiste à modifier le code noyau pour remplacer les instructions non virtualisables par des appels au moniteur de machines virtuelles. En remplaçant un bloc d'instructions sensibles par un seul appel à l'hyperviseur, il est possible de simplifier le code d'émulation du moniteur de machines virtuelles et de diminuer le nombre d'appels qui y est effectué obtenant ainsi un gain de performances.

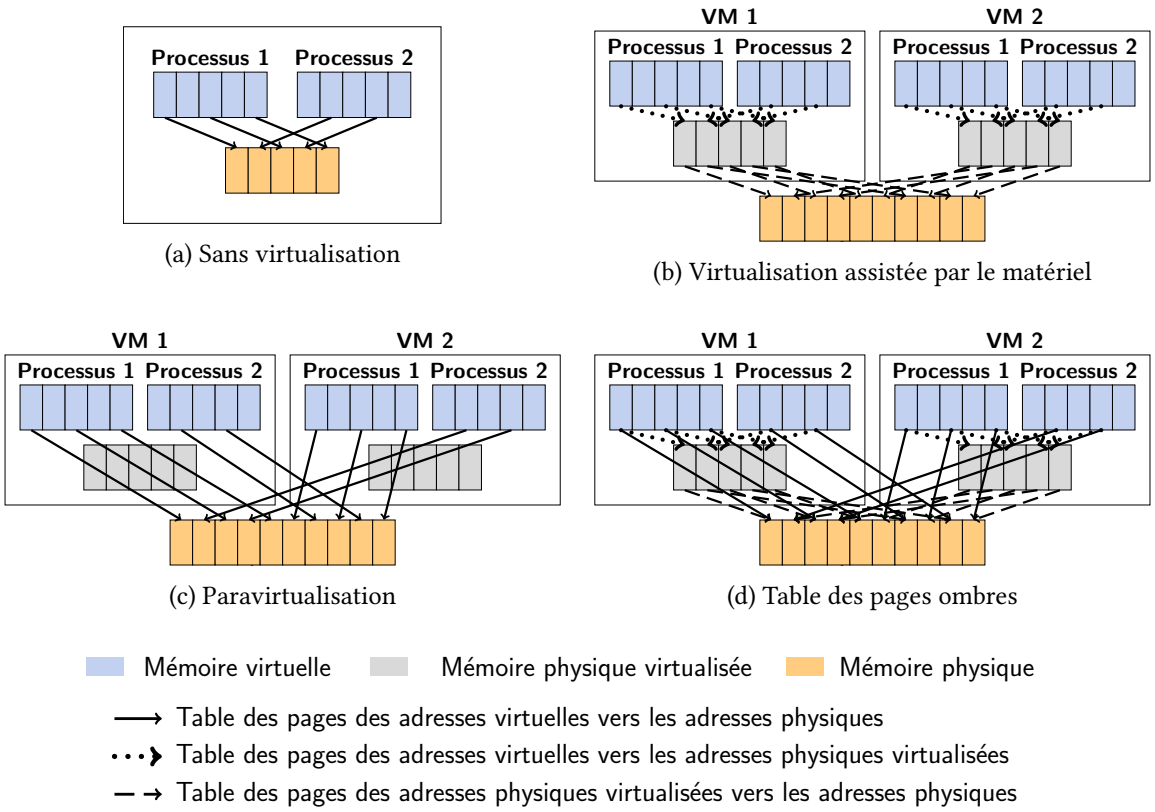
Enfin, la technique de **virtualisation assistée par le matériel** rajoute un mode au processeur afin de détecter l'ensemble des instructions sensibles. Dans certains cas, le moniteur de machines virtuelles peut configurer le matériel pour qu'il émule, en effectuant une action configurée par l'hyperviseur, certaines instructions sensibles diminuant ainsi la fréquence des exceptions.

2.1.2.2 Mémoire

La mémoire est le deuxième composant le plus critique, après le processeur, qui doit être virtualisé. Le but est de répartir des zones de la mémoire physique entre les différentes machines virtuelles. La virtualisation de la mémoire nécessite d'adresser trois défis que sont la traduction des adresses, le partage de l'espace d'adressage virtuel et l'isolation spatiale.

Traduction des adresses La virtualisation du système mémoire utilise un mécanisme de mémoire virtuelle paginée qui est largement utilisé par les architectures matérielles et les systèmes d'exploitation modernes (Figure 2.2a). L'espace d'adressage virtuel est découpé en zones de mêmes tailles appelées pages. Le système d'exploitation maintient des tables, nommées **table des pages**, de correspondance entre les **pages virtuelles** et les **pages physiques** qui sont utilisées par l'unité de gestion de la mémoire pour traduire les adresses virtuelles en adresses physiques ($\longrightarrow$).

L'exécution de plusieurs machines virtuelles sur un unique matériel requiert un niveau de traduction supplémentaire (Figure 2.2b). Désormais, l'hyperviseur possède le contrôle des ressources mémoire de la plate-forme. Les adresses virtuelles sont converties en adresses **physiques virtualisées** en utilisant les tables des pages configurées par l'OS invité, puis les adresses



(a) Sans virtualisation

(b) Virtualisation assistée par le matériel

(c) Paravirtualisation

(d) Table des pages ombres

FIGURE 2.2 – Mémoire paginée

physiques virtualisées sont converties en adresses physiques en utilisant une configuration effectuée par l'hyperviseur.

La technique de **virtualisation assistée par le matériel** (Figure 2.2b) rajoute un niveau de traduction supplémentaire. L'OS invité configure la table des pages effectuant la traduction des adresses virtuelles en adresses physiques virtualisées (⇢) et l'hyperviseur configure une deuxième table des pages associant les adresses physiques virtualisées aux adresses physiques (⇢). A l'exécution, le matériel effectue la double traduction.

Avec la **paravirtualisation** (Figure 2.2c) le système d'exploitation invité est modifié pour qu'il configure les tables des pages de manière à assurer la correspondance entre les adresses virtuelles et physiques (⟶) sans passer par l'étape d'adressage physique virtualisé, l'hyperviseur ayant un simple rôle de vérification pour assurer que l'OS invité ne tente pas d'accéder à des adresses qui lui sont interdites.

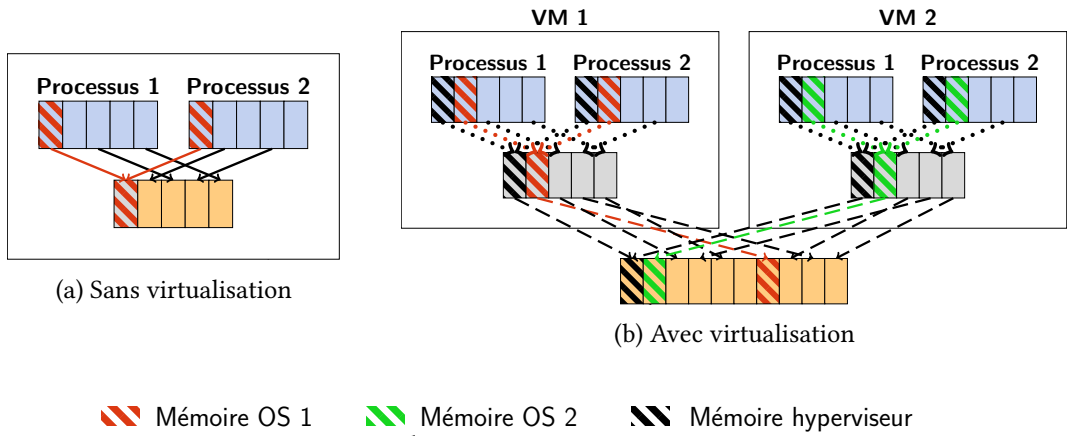
Enfin, par la technologie d'**émulation** (Figure 2.2d), l'hyperviseur piège les instructions sensibles que l'OS invité utilise pour configurer la *MMU* afin de détecter les correspondances des adresses virtuelles vers les adresses physiques virtualisées (⇢). Dans une deuxième étape, l'hyperviseur, effectue la traduction entre les adresses physiques virtualisées et les adresses physiques (⇢) afin de remplir une **table des pages ombres**, utilisée par le matériel, pour traduire directement les adresses virtuelles vers les adresses physiques (⟶).

Maintenant que nous avons vu les différentes solutions proposées pour gérer le niveau additionnel de traduction des adresses requis par la virtualisation, nous allons voir les mécanismes utilisés pour réserver une portion de l'espace d'adressage virtuel au moniteur de machines virtuelles.

Partage de l'espace d'adressage virtuel Les systèmes d'exploitation peuvent utiliser l'ensemble de l'espace d'adressage virtuel du processeur qui est généralement découpé en deux parties : une partie utilisée par le système d'exploitation et une partie utilisée par les processus (Figure 2.3a).

Dans une architecture virtualisée, l'hyperviseur doit pouvoir réserver une portion de l'espace d'adressage virtuel utilisé par le système invité pour contenir, à minima, les structures de contrôle qui gèrent les transitions entre les systèmes invités et l'hyperviseur (Figure 2.3b).

La virtualisation assistée par le matériel permet de résoudre ce problème en utilisant, par exemple, des techniques de **segmentation mémoire** [23]. Il est également possible d'utiliser une technique d'émulation où l'hyperviseur protège les pages virtuelles qu'il s'est attribuées pour détecter et émuler tout accès effectué par les systèmes invités. Enfin, la technique de paravirtualisation implique de modifier les systèmes d'exploitation invités pour s'assurer que les pages virtuelles réservées pour l'hyperviseur ne soient pas utilisées par les systèmes invités supprimant ainsi l'étape d'émulation.



(a) Sans virtualisation

(b) Avec virtualisation

FIGURE 2.3 – Partage des droits mémoire

Isolation spatiale Pour des raisons de sécurité et de sûreté, les systèmes d'exploitation modernes utilisent un mécanisme de protection matérielle pour isoler la mémoire utilisée par les processus de celle utilisée par le système d'exploitation. Les pages mémoires peuvent être dotées d'un niveau de droits corrélé au mode d'exécution du processeur. Dans la figure 2.3a, les pages mémoires du système d'exploitation sont dotées de droits privilégiés et ne peuvent donc être accédées par les processus qui s'exécutent en mode utilisateur.

L'utilisation d'un hyperviseur requiert un niveau de droits supplémentaire (Figure 2.3b). Les pages mémoires utilisées par l'hyperviseur ne doivent pas pouvoir être accédées par les

systèmes d'exploitation invités dont les pages mémoires doivent également être isolées des processus. Lorsque le matériel ne permet pas l'utilisation d'au moins trois niveaux de privilèges différents, les systèmes d'exploitation invités s'exécutent alors avec le même niveau de droits que les processus ce qui introduit des failles de sécurité.

2.1.2.3 Interruptions

Les interruptions et les exceptions sont des événements issus de causes externes (Interruption horloges, ...) ou internes (Exécution d'une instruction privilégiée en mode utilisateur, ...) au programme qui provoquent un déroutement du fil d'exécution vers un autre programme nommé service d'interruption.

Dans une architecture non virtualisée, les systèmes d'exploitation enregistrent leur propre gestionnaire d'interruptions auprès du matériel. Dans une architecture virtualisée l'hyperviseur enregistre son propre gestionnaire d'interruptions et redirige les interruptions vers les gestionnaires d'interruptions des systèmes invités, lorsque ceux-ci sont ordonnancés et que les interruptions ne sont pas masquées.

Pour diminuer le surcoût, en termes de performances, provoqués par l'indirection supplémentaire, la paravirtualisation autorise l'OS invité à enregistrer certains gestionnaires d'interruptions directement auprès du matériel, l'hyperviseur assurant une vérification de la routine d'interruptions pour vérifier que le système invité ne tente pas d'usurper des privilèges.

La virtualisation par le matériel permet d'obtenir les mêmes performances que celles obtenues avec un système paravirtualisé tout en diminuant la complexité d'implémentation dans le moniteur de machines virtuelles.

2.1.2.4 Périphériques

Outre le CPU et la mémoire, les périphériques matériels (Cartes réseaux, GPU, écrans, ...) doivent également être virtualisés. Les contraintes de performances et les modes de fonctionnement qui dépendent du type de composants virtualisés, couplés à l'hétérogénéité du matériel utilisé résulte en une importe complexité de mise en œuvre.

L'émulation peut être utilisée afin de partager les périphériques entre les différents systèmes invités. L'hyperviseur présente, à chaque machine virtuelle, un ensemble de périphériques virtuels qui émulent des périphériques matériels largement utilisés. Les requêtes effectuées par les machines virtuelles sont multiplexées et transmises aux périphériques matériels qui sont gérés par l'hyperviseur. Cette technologie appliquée en l'état aux composants complexes que sont les GPU résulte en de faibles performances.

La **paravirtualisation** repose sur la fourniture, aux systèmes d'exploitation invités, de pilotes modifiés qui effectuent des appels vers l'hyperviseur ce qui permet d'obtenir des gains de performances tout en réduisant la complexité de l'émulation. L'api `VirtIO` [111] a été portée au sein du noyau Linux pour faciliter l'interfaçage entre les hyperviseurs et les systèmes virtualisés. Néanmoins, la mise en œuvre d'une telle stratégie sur des composants matériels tels

que le GPU, dont les pilotes mettent en œuvre une importante pile logicielle (librairie Open GL), est complexe.

La méthode dite du ***pass-through*** permet de garantir de bonnes performances d'accès, au détriment du partage de ladite ressource, en dédiant en exclusivité une ressource à une machine virtuelle. L'hyperviseur va alors allouer les ressources matérielles du périphérique directement au système d'exploitation invité ce qui nécessite, notamment, la présence de matériel spécifique, tel qu'une **IO-MMU**, pour continuer à assurer l'isolation spatiale.

Des supports matériels ont donc été ajoutés pour augmenter les performances aussi bien dans le domaine des réseaux [34] que des processeurs graphiques [130, 132]. Le but de ces technologies est de réduire voir d'éliminer totalement les appels au moniteur de machines virtuelles en les remplaçant par des appels au matériel, garantissant ainsi de bonnes performances sans modifications logicielles dans les systèmes invités. Ces supports matériels ne sont pas actuellement disponibles dans la carte que nous avons utilisée pour effectuer nos travaux en avance de phase. Cependant, les feuilles de routes des fondeurs laissent à penser qu'elles seront ajoutées dans les futures architectures proposées pour le multimédia.

2.1.3 Virtualisation des systèmes embarqués

La technologie de virtualisation a initialement été industrialisée dans le monde des centres de données. Récemment, elle s'est diffusée au domaine de l'informatique embarquée dont les contraintes et les exigences sont fondamentalement différentes de celles couramment mises en œuvre dans les hyperviseurs de l'informatique des serveurs et des applicatifs.

Le cahier des charges des hyperviseurs embarqués contient des exigences [112] en termes d'embarquabilité et d'empreinte mémoire qui doit rester suffisamment petite afin que le code de l'hyperviseur tienne dans la mémoire des systèmes embarqués. L'impact de l'hyperviseur, tant sur les ressources matérielles de la plate-forme que sur les performances temps réel, doit également être minimisé. Le moniteur de machines virtuelles doit en plus être capable de virtualiser des systèmes d'exploitation temps réel en fournissant une forte isolation spatiale et temporelle. Enfin, les architectures couramment utilisées dans les domaines embarqués (ARM, PowerPC) doivent être supportées.

Nous pouvons observer que les fournisseurs d'hyperviseurs pour le domaine de l'embarqué [32, 63, 104, 140] utilisent massivement des hyperviseurs de type 1. Ces hyperviseurs s'exécutent directement sur le matériel de la machine hôte ce qui entraîne une réduction du volume de code déployé sur l'hyperviseur. Cette réduction s'accompagne de gains en terme d'embarquabilité, l'empreinte mémoire de l'hyperviseur au sein du matériel étant réduite, ce qui permet la mise en œuvre de techniques de vérification formelle [98] et de certifications [63, 104] et une réduction de la *Trusted computing base* [123] importante pour la mise en pratique de politiques de sécurité. En outre, le concept d'hyperviseur de type 1 partage des notions communes avec les micro-noyaux [60, 123]. Les fournisseurs d'hyperviseurs [63, 98, 104] peuvent donc réutiliser une partie des concepts qu'ils ont déjà développés dans leurs produits préexistants.

Les acteurs du monde de l'embarqué ont eu tendance, dans les premières solutions de virtualisation, à recourir aux techniques de paravirtualisation au lieu des techniques d'émulation

pour assurer de bonnes performances tout en garantissant une simplicité d'implémentation [60]. L'arrivée de la virtualisation matérielle dans les processeurs embarqués permet maintenant d'augmenter les performances des hyperviseurs existants [140] mais aussi d'envisager la virtualisation de systèmes non modifiés.

2.2 Multi-cœurs

L'accroissement des besoins en ressources CPUs provoqué par l'augmentation des prestations clients a contribué à l'émergence des architectures multi-cœurs dans le monde automobile aussi bien dans la partie mécatronique que dans la partie de l'info-divertissement. Si les architectures matérielles [120] du monde de la mécatronique se caractérisent par une plus forte prise en compte des caractéristiques temps réel, leurs performances sont insuffisantes pour exécuter des systèmes d'exploitation d'info-divertissement. La virtualisation de tels systèmes d'exploitation nécessite donc d'utiliser des architectures matérielles de type COTS déjà employées par le domaine du multimédia.

2.2.1 Modèle de programmation multi-cœurs sur architectures virtualisées



(a) Asymmetric Multi Processing

(b) Symmetric Multi Processing

FIGURE 2.4 – Modèle de programmation multi-cœurs

Deux modèles de programmation peuvent être utilisés pour exploiter les calculateurs multi-cœurs [41].

Dans le modèle de programmation **Asymmetric Multi Processing** (Figure 2.4a) chaque cœur ordonnance une instance distincte d'un système d'exploitation, chaque système s'exécutant de la même manière que si il était exécuté sur un seul CPU mono-cœur. Les systèmes d'exploitation peuvent communiquer entre eux de manière faiblement couplée, par exemple en utilisant des techniques de mémoires partagées. Avec le modèle de programmation **Symmetric Multi Processing** (Figure 2.4b), tous les cœurs sont contrôlés par un unique système d'exploitation qui les répartit entre les fils d'exécutions de l'application qui s'exécute. Les cœurs interagissent de manière fortement couplée en utilisant des verrous pour se synchroniser (Cohérence des caches).

Les avantages du modèle de programmation **AMP** résident dans la possibilité de réutiliser des systèmes d'exploitation patrimoniaux mono-cœurs et d'exécuter des applications qui, par nature, ne sont pas parallélisables. En outre, l'exécution en parallèle de deux systèmes d'exploitation permet d'assurer une bonne réactivité. Le modèle de programmation **SMP** permet

d'amener les bénéfices de la programmation multi-cœurs au niveau des applicatifs et autorise également une connaissance plus fine des ressources matérielles qui sont accédées concurremment. En outre, le modèle de programmation SMP ne nécessite pas la mise en œuvre de mécanismes de synchronisation inter-partitions, des mécanismes de synchronisation devant néanmoins être déployés par le système d'exploitation au niveau applicatif.

2.2.2 Multi-cœurs et contraintes temps réel

Nous allons maintenant étudier l'incidence des architectures multi-cœurs sur les contraintes temps réel. Dans une première partie, nous définirons le concept de « composabilité », utilisé pour garantir le respect des contraintes temps réel dans les architectures opérationnelles fédérées et intégrées, pour ensuite illustrer la remise en cause de ce principe dans les architectures multi-cœurs.

2.2.2.1 Prédictibilité et « composabilité »

La prédictibilité est une des contraintes majeures des systèmes temps réel qui doivent garantir, en toute circonstance, le respect des échéances temporelles des différentes tâches exécutées. La technique de calcul de pire temps d'exécution est utilisée pour effectuer des analyses d'ordonnançabilité afin de démontrer le respect des échéances des tâches ordonnancées au sein d'un système temps réel.

La mise en œuvre de telles techniques devient particulièrement complexe lorsque des dépendances logicielles (Synchronisation inter-tâches ...) ou matérielles (Accès concurrent à une même ressource partagée ...) existent entre les différentes tâches ordonnancées. Le calcul du *WCET* d'une tâche dépend alors des temps d'exécution des autres tâches exécutées.

Des solutions de partitionnement ont été proposées pour résoudre ce problème. Il s'agit de découper le système à vérifier en de multiples sous-parties, indépendantes les unes des autres, qui peuvent être analysées plus facilement que si une analyse était réalisée sur l'ensemble du système. Le principe de « **composabilité** » formalise un tel comportent : un système est dit **composable** si le comportement temporel et fonctionnel d'une application est le même, indépendamment du fait qu'il y ait ou non d'autres applications exécutées sur le système.

Ce principe, qui suppose l'élimination totale des interférences entre applications, est partiellement respecté, par construction, dans les architectures opérationnelles fédérées, chaque prestation client disposant de son propre calculateur, la seule ressource partagée entre les fonctionnalités étant le réseau. L'architecture intégrée AUTOSAR [144] se réclame également du principe de « composabilité » où des composants logiciels issus de multiples fournisseurs sont testés et validés séparément puis intégrés au sein d'un même calculateur. Des politiques d'allocation des ressources matérielles par ordonnancement sont donc mises en œuvre pour assigner lesdites ressources aux composants logiciels.

Les architectures virtualisées embarquées qui mettent en œuvre des systèmes d'exploitation temps réel doivent donc assurer le principe de « composabilité » au moins pour les systèmes d'exploitation temps réels virtualisés. En effet, il paraît inconcevable que les systèmes d'ex-

ploitation d'info-divertissement puissent temporellement perturber les applications exécutées dans le système temps réel.

2.2.2.2 Interférences

La mise en œuvre du principe de « composabilité » dépend de la capacité des architectures matérielles à fournir un système prédictible. Les architectures multi-cœurs modernes apportent des sources d'imprédictibilités additionnelles par rapport à celles des architectures mono-cœurs classiques (Section 1.1.1.3). En effet, si l'on peut dédier des unités de calculs aux différentes applications exécutées en parallèle, il reste à traiter le problème des accès concurrents aux ressources partagées (pour des raisons de coûts, d'énergie, de taille, de communications inter-cœurs). Ainsi, chaque accès effectué par un cœur à une ressource partagée est potentiellement mis en concurrence avec celles effectuées par les autres cœurs. L'arbitrage est alors fait de manière implicite par le matériel, ce qui entraîne une augmentation non déterministe des temps d'exécution des applications [30]. Kotaba et al [74] ont effectué un inventaire, que nous avons repris dans le tableau 2.1, des ressources matérielles, spécifiques aux architectures multi-cœurs modernes de type COTS, qui ajoutent des sources d'interférences temporelles additionnelles par rapport aux architectures mono-cœur.

Le problème des accès concurrents à des ressources partagées devient particulièrement critique lorsque l'on aborde la ressource mémoire. En effet, la latence et la bande passante est un des facteurs majeurs qui limite la performance des processeurs. Les fondeurs ont donc mis en place une hiérarchie de mémoires, partagée entre tous les processeurs, pour essayer d'outrepasser ces limitations. Si les techniques de virtualisation de la mémoire (Section 2.1.2.2) permettent d'assurer l'isolation spatiale, la plupart des architectures matérielles de type COTS ne fournissent pas de mécanismes permettant d'assurer l'isolation temporelle nécessaire au respect du principe de « composabilité ».

Par exemple, dans la figure 2.5, où un système d'exploitation multimédia est ordonnancé en parallèle d'un système d'exploitation temps réel, les temps d'exécution du système temps réel, dépendent des accès mémoire effectués par les applications ordonnancées sur ledit système, mais aussi des accès mémoires effectués par le système multimédia. Notons que la problématique de contention au niveau du système mémoire s'applique également au modèle de programmation SMP. Le système d'exploitation temps réel multi-cœurs doit assurer le respect du principe de « composabilité » entre les fils d'exécutions ordonnancées sur plusieurs cœurs.

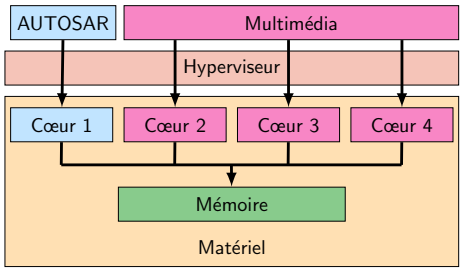


Figure 2.5 – Cas d'utilisation AMP

Ressource partagée	Impacts
Bus système	Ralentissement lorsque des accès concurrents émis par les cœurs, périphériques (DMA) et les protocoles de cohérence outrepassent la capacité du matériel à les servir en parallèle et doivent donc être traités séquentiellement
Passerelle d'interconnexion	Contention générée par les accès effectués par les bus connectés à la passerelle
Bus mémoire et contrôleur	Ralentissement lorsque des accès concurrents dépassent la capacité du matériel à servir en parallèle les accès
Mémoire DRAM	Accès entrelacés provoquant des dépendances entre les données accédées par différents cœurs
Cache partagé	Éviction de lignes du cache d'une tâche par une autre Capacité du cache, à traiter des accès en parallèle, limitée ⟶ Contention et ralentissement Protocole de cohérence des données entraînant un ralentissement des accès au cache
Cache locaux	Protocole de cohérence des données entraînant un ralentissement des accès au cache
TLBs	Protocole de cohérence des données entraînant un ralentissement des accès au cache
Périphériques	Coût des mécanismes de synchronisation pour assurer le multiplexage des accès aux périphériques Présence de contention : des accès parallèles à un même périphérique doivent être traités séquentiellement
Pipeline	Présence de contention lorsque des *threads* partagent un même cœur.
Unité de calculs (GPU)	Présence de contention lorsque des applications partagent une même unité de calcul.

Table 2.1 – Effets indésirables affectant le déterminisme temporel des composants partagés dans les architectures multi-cœurs [74]

La présence de contention mémoire au sein de cette hiérarchie peut entraîner une importante dégradation des performances des cœurs et une augmentation de l'imprédictibilité du système. Nous étudierons dans le chapitre 4 de cette thèse, comment les problèmes de contention, au niveau du système mémoire d'une carte embarquée, se traduisent par une dégradation de performances qui peuvent remettre en cause le principe de « composabilité ». Pour mieux comprendre ce phénomène il est nécessaire d'introduire un certain nombre de concepts et de mécanismes liés à la gestion de la mémoire. Nous allons donc, dans la partie suivante, décrire l'ensemble des composants électroniques présents au sein de la hiérarchie mémoire.

2.3 Hiérarchie mémoire

La **mémoire primaire**, nommée aussi **mémoire principale**, est un dispositif électronique de stockage de données directement accessible par les composants électroniques consommateurs de mémoire (CPU, périphériques) qui vont générer des requêtes d'accès en lecture ou en écriture pour recevoir ou sauvegarder des données.

Introduite dans les années 1970, la *Random-access memory* s'est imposée pour devenir le standard industriel. Elle peut être représentée sous la forme d'un ensemble de **cellules mémoires** connectées entre elles où chaque cellule contient un bit de données. Actuellement, les deux technologies les plus utilisées pour fabriquer les cellules mémoires sont la SRAM (*Static random access memory*) et la DRAM (*Dynamic random access memory*).

Une cellule de SRAM sauvegarde un bit de données dans deux inverseurs montés tête-bêche, chaque inverseur étant composé de deux transistors le tout étant couplé à deux transistors d'accès aux données.

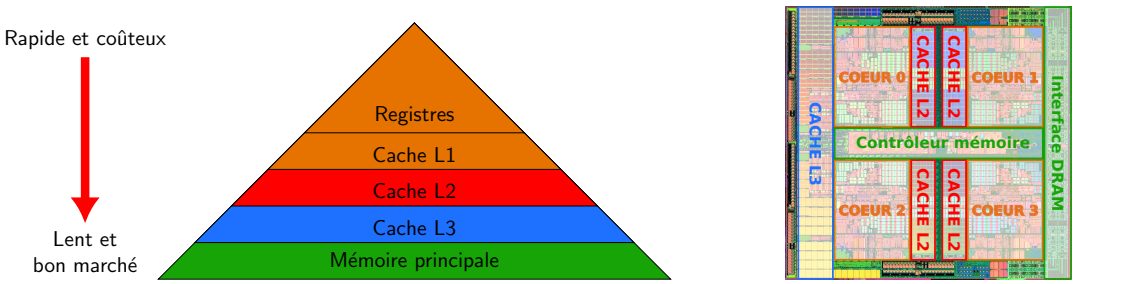
Une cellule de DRAM, elle, sauvegarde un bit de données dans un condensateur qui, selon son état déchargé ou chargé, indique une valeur de stockage de 0 ou de 1. Un transistor supplémentaire permet l'accès à la valeur stockée dans le condensateur. Au cours du temps et selon les opérations effectuées, le condensateur se décharge et perd la valeur sauvegardée. Un rafraîchissement périodique des cellules de la DRAM est donc nécessaire.

L'utilisation d'un condensateur désavantage la DRAM par rapport à la SRAM du point de vue des temps d'accès aux données, du fait du coût des opérations de rafraîchissement, mais l'avantage d'un point de vue coûts de fabrication et densité, le nombre de transistors dans une cellule de DRAM est supérieur à celui présent dans une cellule de SRAM.

2.3.1 Caractéristiques des mémoires

La mémoire peut être caractérisée en utilisant différentes propriétés. La **latence** est utilisée pour définir le temps d'accès à une donnée, la **bande passante** caractérise le volume de données accédées dans une période de temps et la **taille** mémoire détermine la quantité de données pouvant être stockées pour un **coût** donné. Une mémoire idéale aurait une latence et un coût nul, une capacité et une bande passante infinie. Or, les exigences qui caractérisent la mémoire idéale sont antagonistes les unes aux autres. Pour une technologie mémoire donnée, l'accroissement de la taille entraîne une augmentation du temps de décodage de l'adresse, afin de déterminer la localisation de la donnée accédée, ce qui augmente la latence. Pour une capacité de mémoire donnée, la latence est déterminée par le type de technologie utilisé (SRAM, DRAM), plus elle est coûteuse plus la latence est faible. Les gains en bande passante se font par une augmentation de la fréquence mémoire, une augmentation du nombre et de la taille des bus de données ou l'utilisation de nouvelles technologies, le tout se traduisant par une augmentation des coûts.

Pour se rapprocher des caractéristiques des mémoires idéales, les concepteurs des matériels font appel à une **hiérarchie mémoire** (Figure 2.6) qui combine plusieurs niveaux de stockage, progressivement plus gros et lents au fur et à mesure que l'on s'éloigne du processeur. En s'assurant que la plupart des données requises par le processeur sont présents dans les niveaux de cache les plus rapides, les concepteurs réussissent à obtenir des temps d'accès et des débits moyens s'approchant de ceux de la mémoire la plus rapide tout en conservant les capacités mémoires de la mémoire la plus lente.



(a) Pyramide mémoire (b) Processeur AMD barcelona 4 cœurs

FIGURE 2.6 – Hiérarchie mémoire

2.3.2 Caches

Le bon fonctionnement des caches est basé sur la capacité à assurer que les données requises par les consommateurs sont présentes dans le cache. A cet effet le cache se base sur deux caractéristiques qui peuvent être empiriquement observées dans le fonctionnement des programmes : la **localité spatiale** et la **localité temporelle**. La propriété de **localité temporelle**, énoncée par Wilkes en 1965 [145], stipule qu'une donnée récemment accédée à statistiquement plus de chances d'être de nouveau accédée dans un futur proche. En conservant lesdites données dans le niveau de mémoire cache le plus rapide, le processeur s'assure que les hypothétiques et prochains accès à la donnée seront plus rapides. Le principe de **localité spatiale**, introduite par Liptay en 1968 [83] affirme que lorsqu'une donnée est accédée à un moment donnée, alors il y a de fortes chances que les données voisines soient prochainement accédées. L'utilité de cette propriété réside dans le fait que le temps de chargement d'une donnée depuis la mémoire principale vers le cache n'est pas uniforme mais nécessite deux opérations. La première opération, qui a un coût fixe, consiste en un décodage de l'adresse mémoire afin de localiser la donnée accédée dans la mémoire. La deuxième opération de transfert des données est de durée variable selon la taille des données accédées. Il est donc plus rentable de précharger un bloc de données depuis la mémoire principale vers le cache pour profiter de la localité spatiale que d'effectuer plusieurs transferts de plus petites tailles.

2.3.2.1 Accès au cache

Le processeur qui souhaite accéder à une donnée présente en mémoire principale va envoyer une requête au contrôleur du cache de plus bas niveau. Lorsque la donnée est présente dans le cache on parle de **succès de cache** ou de *Cache HIT* et de **défaut de cache** ou *Cache MISS* lorsque la donnée en est absente. Dans le premier cas, la donnée est transférée depuis le cache vers le processeur alors que dans le second cas, la donnée est alors récupérée depuis les caches de plus haut niveaux ou depuis la RAM, copiée dans le cache et transférée vers le processeur.

Une distinction est effectuée entre plusieurs types de défaut de cache :

— Les défauts de cache dits **obligatoires** qui ne peuvent, comme leur nom l'indique, être évités résultent de la première demande effectuée par un processeur pour obtenir une donnée ;

— On parle de défauts de cache **capacitifs** lorsque le cache est plein et que le chargement de nouvelles données entraine une éviction des anciennes ;

— Les défauts de cache **conflictuels** sont provoqués lorsque deux données distinctes sont enregistrées dans la même ligne de cache, le chargement d'une donnée évinçant l'autre ;

— Les défauts de **cohérence** sont provoqués lorsque des lignes du cache sont invalidées pour conserver la cohérence entre les différents caches d'un multiprocesseur.

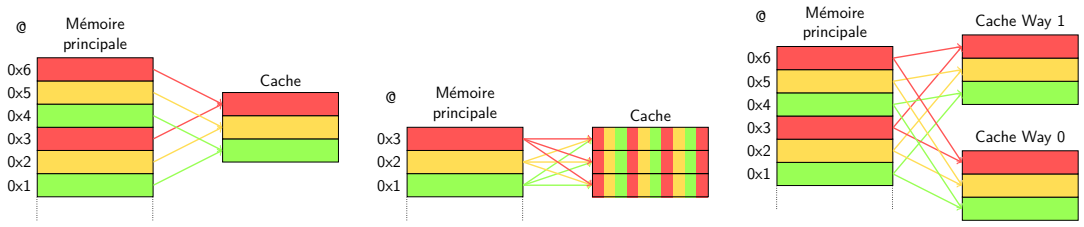
2.3.2.2 Organisation

Un cache peut être représenté sous la forme d'un vecteur de **lignes** mémoire, appelées aussi **blocs**, où chaque ligne peut contenir une copie d'un bloc de données présentes dans la mémoire principale. Une ligne de cache représente la plus petite unité de données utilisée pour communiquer avec la mémoire de niveau supérieur. A chaque fois qu'une donnée est transférée depuis le cache vers la mémoire de niveau supérieur, la quantité minimal de donnée transférée est donc égale à une ligne cache. Une ligne de cache est découpée en mots mémoires. Un mot mémoire représente le plus petit élément de données qui peut être transféré entre le processeur et la mémoire cache.

A chaque ligne de cache est associé un ensemble de métadonnées comprenant le *tag*, partie de l'adresse mémoire utilisée pour identifier quel bloc mémoire est présent dans une ligne de cache, et des bits pour indiquer l'état de chaque ligne.

2.3.2.3 Politiques de correspondance

Différentes politiques de correspondance peuvent être utilisées pour définir dans quelles lignes de cache un bloc de la mémoire principale peut être placé.



(a) Cache à correspondance préétablie (b) Cache totalement associatif (c) Cache partiellement associatif

FIGURE 2.7 – Placement des données

Dans les caches à correspondance préétablie, nommés aussi *Direct-Mapped* cache, chaque bloc de données de la mémoire principale peut être placé dans une et une seule ligne de la mémoire cache (Figure 2.7a). La figure 2.8a illustre les étapes d'accès à un tel cache.

1. L'adresse mémoire de la donnée accédée est découpée en quatre parties.
2. Les bits de la partie **index** sont utilisés pour sélectionner la ligne du cache dans laquelle la donnée est éventuellement présente.

3. Le tag associé à la ligne sélectionnée est comparé avec la partie tag de l'adresse de la donnée accédée pour vérifier que le bloc de données présent dans le cache est bien celui que l'on souhaite accéder.
4. Le processeur vérifie ensuite que la ligne de cache est valide en utilisant les bits d'état des métadonnées.
5. Si les étapes 3) et 4) sont validées, la donnée est présente dans le cache (Cache HIT).
6. Le mot mémoire demandé est alors sélectionné au sein de la ligne de cache et transféré au processeur.
7. La partie octet de l'adresse est utilisée pour sélectionner l'octet utilisé au sein du mot mémoire.

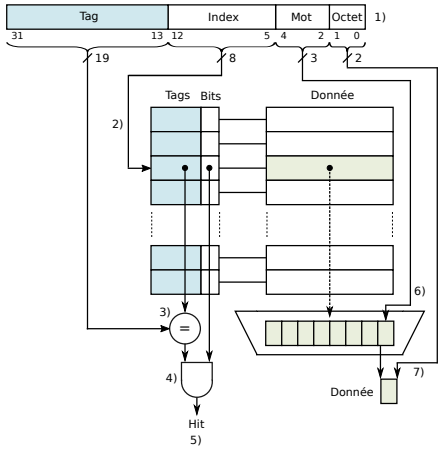
Les avantages de cette politique de placement résident dans sa simplicité de mise en œuvre ce qui se traduit par de faibles coûts matériels et par une faible latence d'accès aux données. En revanche, cette solution a pour défaut d'être à l'origine de nombreux MISS conflictuels.

Dans les caches totalement associatifs, dénommés aussi *fully associative* cache, un bloc de la mémoire principale peut être localisé dans n'importe quelle ligne du cache (Figure 2.7b). Le chargement d'un nouveau bloc dans le cache est de préférence effectué dans les lignes vides ou invalides. Lorsque toutes les lignes du cache sont valides et que des défauts de cache capacitaire se produisent, le cache fait appel à une **politique de remplacement** des données afin de sélectionner le bloc qui va être remplacé dans le cache. Différentes politiques de remplacement des données peuvent être utilisées tel que *FIFO*, *random* ou *Least recent used*. Si la mise en œuvre de cette politique de placement permet de totalement supprimer les défauts de cache dits conflictuels, elle nécessite en revanche, la présence de composants matériel de logique pour rechercher la localisation d'une donnée dans le cache ce qui se traduit par une augmentation des coûts et de la latence d'accès aux données du cache.

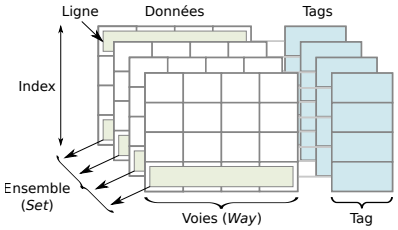
Les caches partiellement associatifs, appelés aussi *Set Associativity* cache, représentent un compromis entre les caches à correspondance préétablie et les caches totalement associatifs. Chaque bloc de données de la mémoire principale peut être associé à un nombre limité de lignes de la mémoire cache définis à l'avance (Figure 2.7c). La figure 2.8b illustre l'architecture d'un cache partiellement associatif. Le cache est découpé en différentes voies (*way*). La sélection de l'ensemble (*set*), dans lequel un bloc de données peut être écrit, est effectuée en utilisant l'index de l'adresse mémoire. Une fois l'ensemble sélectionné, la recherche ou le placement d'une donnée dans ledit ensemble requiert l'utilisation d'une politique de remplacement des données. Le choix du degré d'associativité utilisé dans un cache fait l'objet d'un compromis entre le coût du matériel et l'augmentation des temps de latence qu'entraîne un haut niveau d'associativité et l'augmentation du taux de MISS provoqué par un faible taux d'associativité.

2.3.2.4 Politique de gestion des écritures

La politique de gestion des écritures d'un cache est utilisée pour décider quand est-ce que les opérations d'écritures sont propagées vers la mémoire de niveau supérieure et in fine dans la mémoire principale. Actuellement, deux politiques de gestion des écritures peuvent être utilisées.



(a) Cache à correspondance préétablie d'une taille de 8 kilo octets



(b) Cache partiellement associatif

FIGURE 2.8 – Terminologie des caches

Avec la politique d'exécution à **écriture différée** nommée aussi *write-back*, les écritures sont effectuées dans le cache et leur propagation vers la mémoire de plus bas niveau est retardée jusqu'au moment où le bloc de données est évincé du cache. Ce type de politique permet de diminuer la bande passante mémoire requise, plusieurs modifications sur une ligne de cache déjà chargée n'entraînant pas d'écritures vers les niveaux supérieurs. En revanche, elle nécessite la présence de composants matériels supplémentaire, notamment sous la forme d'un bit *dirty* pour indiquer qu'une ligne de cache a été modifiée.

Dans la politique **d'écriture immédiate** appelée aussi *write-through*, les écritures sont instantanément propagées depuis le cache vers la mémoire principale. Toute la hiérarchie mémoire dispose donc, à tout moment, des mêmes valeurs ce qui simplifie les politiques de cohérence des caches au détriment d'une consommation en bande passante plus importante.

2.3.2.5 Politique d'allocation

La politique d'allocation des données vise à déterminer si une donnée chargée depuis le niveau de cache supérieur doit être copiée dans le cache (*line fill*) ou non. La politique *read-allocate* effectue un *line fill* uniquement lorsqu'un MISS en lecture est effectué. Lorsqu'un MISS en écriture survient, le cache n'est pas modifié et l'écriture est transmise au niveau supérieur.

La politique d'allocation *write-allocate* alloue une ligne de cache sur MISS en lecture ou en écriture. Cette politique est habituellement utilisée en combinaison avec la politique *write-back*.

2.3.2.6 Caches et multi-cœurs

La présence de caches partagés dans les architectures multi-cœurs modernes entraîne trois effets indésirables majeurs :

— **Contention temporelle :** Le nombre d'accès mémoire pouvant être servis en parallèle

par un cache partagé est limité et ne correspond pas forcément au nombre total de requêtes pouvant être émis par l'ensemble des cœurs. Lorsque ce nombre dépasse la capacité du cache à paralléliser les accès, les requêtes mémoire sont traitées séquentiellement, ce qui entraîne une augmentation des latences d'accès. On parle alors de **contention temporelle** sur les caches.

— **Contention spatiale :** Ensuite, la taille du cache partagé effectivement disponible pour une application dépend, à la fois, de la politique de remplacement des données du cache et des écritures effectuées par les autres applications. En effet, ces dernières peuvent évincer hors du cache les données stockées par la première application, on parle alors de **contention spatiale**. Des **interférences inter-cœurs :** se produisent lorsque deux tâches exécutées sur des cœurs différents accèdent à un même cache commun et que les accès d'une des tâches évincent les blocs mémoire de l'autre tâche augmentant ainsi son temps d'exécution. La présence d'interférences inter-cœurs entraîne une augmentation considérable des pires temps d'exécution se traduisant par un surdimensionnement inacceptable des plates-formes matérielles.

— **Coût du protocole de cohérence :** Enfin, les protocoles de gestion de la cohérence des données mis en œuvre, lorsque deux consommateurs accèdent à une zone mémoire partagée, introduisent des ralentissements additionnels aussi bien sur les caches partagés que sur les caches privés des cœurs, des données du cache devant être validées et mises à jour.

La conséquence globale de ces trois effets est une augmentation importante du pire temps d'exécution des applications exécutées sur une telle plate-forme, ces derniers dépendant du comportent des applications co-ordonnancées.

Si l'utilisation des caches, dans un contexte multi-cœurs, pose des problèmes de contention spatiale et temporelle entre les tâches exécutées en parallèle, la présence de mémoire principale de type DRAM amène des complications supplémentaires. En effet, les architectures multi-cœurs modernes ont une structure mémoire principale complexe et non uniforme.

Nous allons voir plus en détail, dans la section suivante, l'agencement de la mémoire principale de type DRAM et les impacts de cette organisation sur la « composabilité » de notre plate-forme matérielle.

2.3.3 Mémoire principale

La mémoire de stockage peut être modélisée sous la forme d'un ensemble de cellules connectées au contrôleur mémoire. Dans une implémentation 1 à N de ce modèle, l'ensemble des cellules sont connectées au contrôleur par un unique bus partagé alors que dans une implémentation N à N, chaque cellule est connectée au contrôleur mémoire par un bus dédié.

La première implémentation est peu coûteuse, un seul bus étant nécessaire, mais résulte en de faibles performances. Les accès à un grand tableau monolithique de cellules sont lents. Il existe une forte latence entre le moment où la requête arrive au tableau de donnée et le moment où le transfert des données commence. De plus les accès mémoires ne peuvent être parallélisés.

La deuxième implémentation diminue la latence d'accès aux données et permet de paralléliser les accès aux cellules pour traiter plusieurs requêtes d'accès mémoire en même temps mais est techniquement impossible et économiquement non viable, le grand nombre de bus requis entraînant une explosion des coûts et des impossibilités de placement-routage.

L'architecture de la DRAM est donc un compromis entre les deux implémentations pour garantir de bonnes performances tout en assurant des coûts maîtrisés.

2.3.3.1 Organisation de la DRAM

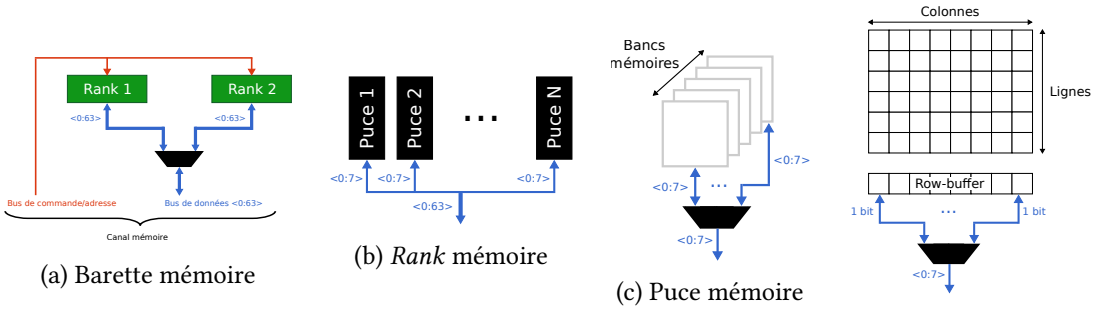
Le système mémoire principal se compose d'un ou de plusieurs contrôleurs mémoires, qui transmettent les requêtes issues des demandeurs (CPU, DMA, caches), vers des barrettes mémoires (*Dual Inline Memory Module*) qui stockent les données.

Un canal de communication (*channel*) connecte de manière exclusive un contrôleur à une ou plusieurs barrettes (Figure 2.9a). Chaque contrôleur peut gérer un ou plusieurs canaux, dont les accès, totalement indépendants, peuvent être effectués en parallèle. Un canal est composé de trois bus, le **bus de commande** (Unidirectionnel), le **bus d'adresse** (Unidirectionnel) et le **bus de données** (Bidirectionnel). Le contrôleur reçoit des requêtes mémoires (Lecture/Écriture de bloc de données) des demandeurs et les décompose en plusieurs **commandes mémoires**. Les commandes sont transférées aux barrettes en utilisant le bus de commande tandis que le bus d'adresse est utilisé pour transférer les adresses des données accédées par la commande, le bus de donnée étant utilisé pour transférer les données résultant des commandes.

Les barrettes sont découpées en sous-ensembles nommés *rank* (Figure 2.9a) qui sont tous connectés au même canal. Les accès parallèles aux *ranks* ne sont pas autorisés, le contrôleur mémoire sélectionne donc le *rank* vers lequel il souhaite envoyer une commande. Un *rank* est composé de plusieurs puces mémoire (Figure 2.9b) qui partagent le même bus de commande et d'adresse mais disposent d'un sous ensemble réservé du bus de données. Une commande émise par le contrôleur mémoire à un *rank* est donc reçue et traitée par l'ensemble des puces dudit *rank* qui transfèrent simultanément les données requises. Une puce est composée de plusieurs **bancs mémoire** (Figure 2.9c) qui contiennent les données. L'ensemble des bancs mémoire d'une puce partagent le même bus de données ce qui interdit tout parallélisme de transfert entre bancs mémoire co-localisés sur une même puce. Un banc peut être représenté sous la forme d'un tableau de cellules mémoire à deux dimensions organisé en ligne et en colonne (Figure 2.9d). Une ligne de cellules supplémentaire nommée *row-buffer* est présente dans chaque banc. Tout accès effectué à une cellule de données d'un banc mémoire passe nécessairement par le chargement de la ligne contenant ladite donnée dans le *row-buffer*. Le *row-buffer* peut donc être vu comme un cache *write-back* d'un banc mémoire, les accès effectués à une ligne déjà présente dans le cache (*row-open*) se traduisant par un *row-hit* sont accélérés par rapport à des accès qui nécessitent de recharger une ligne fermée (*row-close*) et qui se traduisent par un *row-miss*.

2.3.3.2 Latence des accès

Le canal, le *rank*, le banc, la ligne et la colonne dans laquelle une donnée est lue ou écrite, sont déterminés par le contrôleur mémoire à partir des bits de l'adresse physique de la donnée



(a) Barette mémoire

(b) *Rank* mémoire

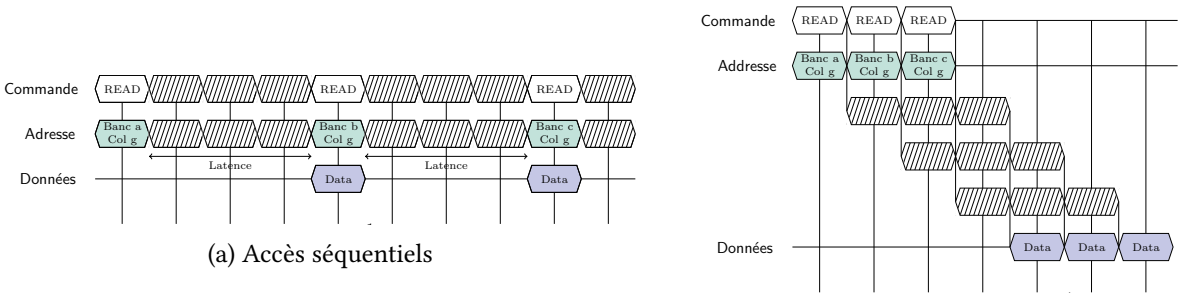
(c) Puce mémoire

(d) Banc mémoire

FIGURE 2.9 – DRAM

accédée.

Entre le moment où la commande atteint le banc mémoire cible et le moment où les données sont transmises sur le bus de données, des contraintes temporelles, nommées aussi *timing*, dues au matériel doivent impérativement être respectées. Les contraintes temporelles et les commandes mémoires sont définies par le standard JEDEC [68]. La figure 2.10a montre l'impact des contraintes temporelles inter-commandes. Les commandes de lecture, sont séquencées ce qui entraîne une sous-utilisation du bus de données et une augmentation de la latence. La technique d'*interleaving* nommée aussi *banking* permet de réduire la latence et d'exploiter le bus de donnée en chevauchant les commandes d'accès mémoire pour recouvrir les temps de latences par des accès au bus (Figure 2.10b). L'utilisation de plusieurs canaux sert le même dessin mais utilise des bus séparés ce qui permet des accès totalement parallèles.



(a) Accès séquentiels

(b) Banking

FIGURE 2.10 – Latences d'accès à la mémoire

On distingue plusieurs cas pathologiques qui augmentent la latence des accès mémoire. Les *banks-conflicts* où deux commandes essayent d'accéder au même banc mémoire entraînant une séquentialisation des commandes. On parle de *row-conflicts*, lorsque deux commandes accèdent à une ligne différente d'un même banc ce qui entraîne un rechargement du *row-buffer* augmentant la latence de la commande.

2.3.3.3 Placement des données

Le choix des bits utilisés par le contrôleur mémoire pour effectuer le placement des données dans les bancs mémoire est tout sauf anodin. Les performances des accès mémoire effectués dépendant du placement des données, il s'agit de minimiser les conflits en répartissant les données sur l'ensemble de la mémoire du système. Plusieurs politiques de placement peuvent être utilisées pour répartir les données au sein des bancs mémoire. La politique de placement de type *cache block interleaving* consiste à répartir les blocs de cache contigus en mémoire sur des bancs mémoire différents et consécutifs permettant ainsi un recouvrement des accès à des blocs contigus. De nombreuses autres politiques de placement de données peuvent également être utilisées (*word interleaving*, *row interleaving*, *cache block interleaving*, *OS page interleaving*).

Le logiciel dispose de moyens de contrôle sur le placement des données dans la RAM. L'unité de gestion mémoire (*Memory Management Unit*) cogérée par le matériel et par le système d'exploitation effectue une traduction des adresses virtuelles en adresses physiques. Les adresses virtuelles sont découpées en deux parties, une partie utilisée pour calculer **l'adresse de page**, la deuxième partie étant utilisée pour calculer le **décalage** au sein de la page. Lors de la conversion des adresses virtuelles en adresses physiques la partie adresse de page de l'adresse virtuelle est convertie par la MMU, qui utilise les tables des pages configurées par le système d'exploitation, tandis que la partie décalage est simplement copiée de l'adresse virtuelle vers l'adresse physique. Le contrôle que le système d'exploitation dispose sur le placement des données dépend donc de l'emplacement des bits de placement et de la taille des pages utilisés. En effet, seul les bits de placement positionnés dans l'adresse de page peuvent faire l'objet d'un contrôle de la part du système d'exploitation.

Actuellement, les systèmes d'exploitation existants perçoivent la mémoire comme une seule ressource uniforme et ne tiennent pas compte de l'architecture physique lors de l'allocation des données.

2.3.3.4 Contrôleur mémoire

Le contrôleur mémoire a pour objectifs d'assurer le fonctionnement correct de la DRAM en respectant les *timings* et en gérant le rafraîchissement des bancs mémoire. Il récupère les **requêtes d'accès mémoire** en lecture ou en écriture depuis les consommateurs et les transforme en **commandes mémoire** qui agissent sur les bancs mémoire. Quatre commandes mémoire sont standardisées par le JDEC :

`ACT` Charge une ligne dans un *row-buffer*

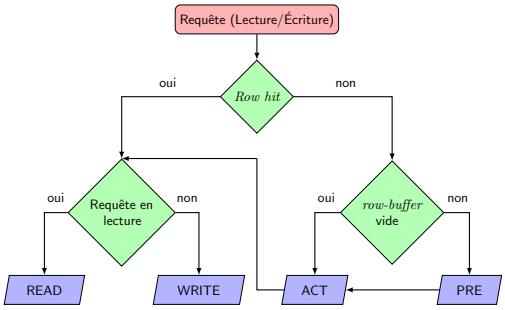
`PRE` Flush une ligne présente dans *row-buffer* dans les cellules de données de la mémoire.

`READ` Lecture d'une donnée depuis un *row-buffer* [1]

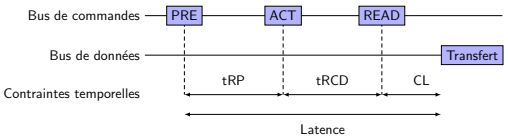
`WRITE` Ecriture d'une donnée dans un *row-buffer* [1]

`REF` Pour préserver les données, une commande de rafraîchissement doit périodiquement être émise sur tous les bancs mémoire. Elle a pour effet secondaire de vider les *row-buffers*.

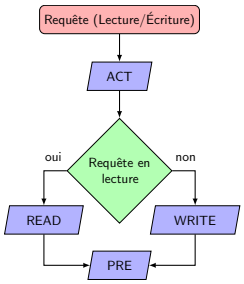
1. Les commandes READ/WRITE sont désignée sous le terme générique de `CAS`.



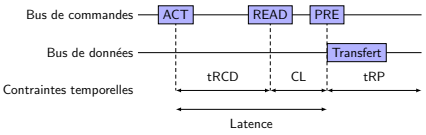
(a) Algorithme de la politique *open-row*



(b) Pire cas de la politique *open-row*



(c) Algorithme de la politique *close-row*



(d) Pire cas de la politique *close-row*

FIGURE 2.11 – Politiques de gestion des *row-buffers*

Deux politiques différentes sont couramment [61] utilisées pour gérer les *row-buffers*.

La politique *open-row* dans laquelle le contrôleur va garder une ligne ouverte après un accès. La ligne est fermée uniquement lorsque un rafraîchissement ou un *row-miss* se produit (Figure 2.11a). Cette politique favorise les applications avec de fortes localités spatiale et temporelle et en conséquence ayant un fort taux de *row-hit*. En revanche, elle entraîne de fortes latences pour les applications qui souffrent de nombreux *row-conflict*, chaque *miss* entraînant une possible commande `PRE` suivie d'une commande `ACT` et d'une commande `CAS` (Figure 2.11b).

Dans la politique *close-row*, le contrôleur va clore la ligne accédée après une commande `CAS` si aucune requête en attente vers la même ligne n'est dans le buffer du contrôleur (Figure 2.11c). Cette politique supprime les *row-conflit* et à un meilleur pire cas (Figure 2.11d) qu'une politique *open-row*. Mais elle augmente la latence pour la majorité des requêtes mémoire.

Les contrôleurs mémoires utilisent des politiques de ré-ordonnancement des commandes d'accès mémoire pour atteindre leurs objectifs. La politique FR-FCFS, couramment utilisée, cherche à maximiser le débit mémoire total. Elle favorise tout d'abord les commandes qui vont résulter en un *row-hit* puis, dans un deuxième temps, favorise les requêtes les plus anciennes par rapport aux requêtes les plus jeunes.

2.3.3.5 Mémoire principale et multi-cœurs

L'utilisation de mémoire principale de type DRAM engendre des interférences temporelles entre les multiples cœurs de la plate-forme matérielle.

Tout d'abord la capacité du contrôleur mémoire à servir en parallèle plusieurs accès vers le système, mémoire est limitée et dépend, notamment, du nombre de canaux mémoires présents au sein du matériel. Lorsque de multiples requêtes mémoire se présentent, le contrôleur met en œuvre une politique d'arbitrage pour sélectionner les accès qui vont être prioritairement effectués. De la politique d'arbitrage utilisée par le contrôleur mémoire dépend la prédictibilité de la plate-forme. Si l'utilisation d'une politique à temps partagé permet de borner les latences mémoire, les contrôleurs de type COTS ont tendance à utiliser des politiques visant à maximiser le débit au détriment de la prédictibilité.

La politique de placement des données utilisée par le contrôleur joue également un grand rôle. Si les politiques d'entrelacements permettent d'exploiter au maximum la bande passante mémoire elles introduisent des dépendances cachées entre les multiples consommateurs qui voient leurs données co-localisées (Cas pathologiques de *banks-conflicts* et de *row-conflicts*).

Enfin, l'utilisation d'une politique de gestion des *row-buffers* de type *open-row* garantit de meilleures performances au détriment des pire temps d'exécutions qui peuvent se produire.

Chapitre 3

Etat de l'art

Sommaire

3.1 Approche par composants matériels 46
3.1.1 Classification des solutions 46
3.1.2 Caches processeurs 47
3.1.3 Contrôleur mémoire 55
3.1.4 Mémoire principale 58
3.1.5 Bus matériels 61
3.1.6 Conclusion 63
3.2 Approches globales 63
3.2.1 Contrôle des accès mémoire 64
3.2.2 Régulation des accès mémoire 69
3.3 Conclusion 73

Dans le chapitre précédent, nous avons vu les problèmes posés par la mise en œuvre d'une architecture opérationnelle virtualisée regroupant, sur une même plate-forme matérielle multi-cœurs, des systèmes d'exploitation temps réel et multimédia. En effet la hiérarchie mémoire de telles plates-formes matérielles étant partagée, des problèmes d'interférences et de contention sur le système mémoire peuvent survenir et remettre en cause le principe de « composabilité » sur lequel repose la mise en œuvre des architectures intégrées et virtualisées. Nous allons maintenant, dans ce chapitre, étudier les différentes approches de la littérature qui ont été publiées pour adresser les problèmes d'interférences inter-cœurs.

Dans une première partie, nous allons détailler, pour chaque composant matériel de la hiérarchie mémoire partagé entre les cœurs, les différentes solutions qui ont été proposées pour gérer les interférences. Ensuite, dans une deuxième section, nous allons aborder les approches qui ont été mises en œuvre au niveau global pour contraindre l'utilisation des ressources partagées afin de borner les interférences générées par des accès concurrents.

3.1 Approche par composants matériels

Les problèmes de contention peuvent survenir à de multiples endroits de la hiérarchie mémoire que nous avons décrite dans la section 2.3. Dans cette section, nous allons détailler, pour chacun des composants matériels de la hiérarchie mémoire, les différents mécanismes qui ont été publiés afin de réduire, d'éliminer ou de borner les interférences engendrées par la contention au niveau du système mémoire.

3.1.1 Classification des solutions

Afin de mettre en perspective les différentes approches, nous avons effectué une classification des multiples solutions en trois familles différentes qui peuvent s'appliquer de manière transverse aux multiples composants électroniques de la hiérarchie mémoire.

1. Les méthodologies basées sur des **calculs de pire temps d'exécution** visent à caractériser et à borner le temps maximal d'exécution des logiciels afin de dimensionner la plate-forme matérielle de telle sorte à ce que les contraintes temps réel soient toujours respectées. La mise en œuvre d'une telle approche, sur une plate-forme matérielle multi-cœurs, nécessite de caractériser le surcoût temporel provoqué par les interférences générées par les accès concurrents.
2. Des **méthodes logicielles**, visant à réduire ou à éliminer les interférences par une configuration fine du matériel, ont également été mises en œuvre. Elles peuvent être appliquées aussi bien sur des architectures matérielles existantes que sur des architectures modifiées et permettent de réduire les pires temps d'exécutions calculés.
3. Enfin, les approches de **re-conception du matériel** apparaissent être les plus adaptées pour résoudre, par construction, les problèmes d'interférences puisqu'elles intègrent dans leur cahier des charges des exigences en termes de prédictibilité et de « composabilité ». Elles ont donc fait l'objet de nombreuses publications étudiant aussi bien la conception de nouvelles architectures matérielles MERASA [134], JOP [115], ComSOC [50, 56], ACROSS [37], PRET [36, 85], TCREST [116] que la re-conception partielle de composants (Contrôleur mémoire, caches, ...).

Le tableau 3.1 contient une synthèse des différentes solutions de l'état de l'art que nous allons détailler dans la suite de cette section.

Composants	Approche		Références
Caches	Pire temps d'exécution		[147], [57], [81], [79], [54]
	Matérielles	Partitionnement *index-based*	[84], [122], [67], [108], [29]
		Partitionnement *way-based*	[28], [126], [107], [80]
		Verrouillage	[12], [114]
	Logicielles-matérielles	Partitionnement *index-based*	[125]
		Partitionnement *way-based*	[135], [93]
		Verrouillage	[125], [87]
	Logicielles	Partitionnement *index-based*	[127], [82], [151], [21], [53], [72], [141]
Contrôleur mémoire	Pire temps d'exécution		[101], [146], [73]
	Matérielles		[2], [109], [99], [75]
Mémoire principale	Logicielles-matérielles		[94], [124]
	Logicielles		[86], [149]
Bus systèmes	Pire temps d'exécution		[143], [110], [27], [110], [69], [27], [110], [27], [26]
	Matérielles		[56], [50], [37]

TABLE 3.1 – Synthèse des solutions de l'état de l'art classifiées par famille d'approche

3.1.2 Caches processeurs

Nous allons maintenant étudier le premier élément de la hiérarchie mémoire, à savoir, les caches des processeurs. Ces composants matériels, utilisés de manière transparente par les programmeurs, appliquent des heuristiques, basées sur les propriétés de localité spatiale et temporelle, pour garder les données les plus fréquemment accédées en cache réduisant ainsi les temps d'exécutions moyens des applications. L'utilisation de ces heuristiques introduit une importante variabilité dans les temps d'accès à la mémoire, plusieurs accès mémoire effectués à un même emplacement par une même instruction se traduisant par des latences d'accès différentes selon l'état du cache.

Nous avons vu, dans la section 2.3.2.6, que, dans une architecture multi-cœurs, cette variabilité est exacerbée, l'état d'un cache partagé dépendant des accès effectués par les tâches exécutées en parallèle.

En suivant notre classification, nous allons dans une première partie, évoquer les différents travaux qui ont été effectués pour produire des analyses de pire temps d'exécution sur des caches partagés au sein d'architectures multi-cœurs. Ensuite, dans les deux sections suivantes, nous allons détailler les techniques logicielles et matérielles de partitionnement et de verrouillage qui ont été proposées pour établir une isolation temporelle entre les différents com-

posants logiciels utilisateurs de caches partagés. Enfin, dans une dernière partie, nous allons étudier un composant matériel, dénommé *scratchpad*, qui a été proposé comme une alternative plus prédictible pour suppléer aux caches.

3.1.2.1 Calculs de pire temps d'exécutions

Différentes techniques de calcul de pire temps d'exécutions ont été développées pour modéliser le comportement des caches de systèmes monoprocesseurs. Ces techniques ne peuvent, malheureusement, être appliquées telles quelles sur les caches partagés utilisés par les processeurs multi-cœurs, les interférences inter-cœurs additionnelles devant être prises en considération. La prise en compte de ces contraintes supplémentaires est ardue et les auteurs de [125] estiment qu'il sera « difficile sinon impossible de développer une méthode capable de capturer précisément la contention dans un système multi-cœurs utilisant un cache partagé » [1]. En dépit de cette opinion des travaux ont été effectués pour résoudre ce problème.

Yan et al [147] proposent une des premières analyses de pire temps d'exécution s'appliquant sur les caches partagés d'un multi-cœur. Leur architecture matérielle de référence se compose d'un dual-cœur, chaque cœur, ayant deux caches de niveau un d'instruction et de donnée privé, est connecté à un cache d'instruction de niveau deux partagé. Pour simplifier leur analyse, les auteurs ont émis l'hypothèse que le cache L1 de donnée est « parfait ». Leur modèle de tâches se compose de deux *threads*, dont le code est connu par avance, exécutés en parallèle sur les deux cœurs de la machine. Un des *thread* possède des contraintes temps réel et est perturbé par le second fil d'exécution non temps réel, le but des travaux étant de caractériser l'impact temporel du deuxième *thread* sur le premier. L'idée sur laquelle repose leur approche consiste à calculer les pires interférences générées lors des accès aux instructions en utilisant un analyseur statique exécuté sur le **graphe de flot de contrôle** des *threads* ordonnancés en parallèle. Cette analyse permet de calculer l'état de chaque bloc d'instruction des programmes pour déterminer s'il est toujours présent dans le cache L2 (***always HIT***), toujours présent dans le cache L2 sauf une fois, (***always-except-one***) où, dans le cas restant, étant considéré comme n'étant jamais dans le cache (***L2 MISS***). Les auteurs combinent les informations ainsi obtenues pour calculer le pire temps d'exécution final en utilisant la théorie de l'interprétation abstraite.

Hardy et al [57] considèrent que la méthode précédemment proposée par Yan et al ne passe pas à l'échelle, lorsque le nombre de tâches non temps réel en face desquelles une tâche temps réel peut être ordonnancée en parallèle augmente, chaque conflit de cache possible devant être pris en considération. Ils proposent une méthode qui s'applique à une architecture matérielle multi-cœurs disposant de multiples niveaux de caches d'instructions partagés non inclusifs, chaque cache ayant plusieurs niveaux d'associativités gérés avec une politique d'allocation *LRU*. Leur modèle de tâches s'applique à un nombre arbitraire de tâches temps réel mais présuppose qu'une tâche ne migre pas entre deux cœurs au sein d'une activation. Les auteurs observent que certains blocs de mémoire copiés dans le cache, lors de l'occurrence d'un cache MISS ne sont jamais ré-accédés avant leur prochaine éviction. Une méthode d'analyse statique de code est donc utilisée sur le code du programme temps réel pour détecter les occurrences

1. In our opinion, it will be extremely difficult, if not impossible, to develop such a method that can accurately capture the contention.

de blocs mémoire ayant de telles propriétés, afin de les marquer comme non *cachable* au sein du code source. A l'exécution, les blocs ainsi tagués sont chargés depuis la mémoire principale sans être copiés dans le cache (*bypass*) réduisant ainsi, la pollution du cache et contribuant à une baisse des pires temps d'exécutions calculés.

Li et al [81] ont développé une méthode de calcul de WCET s'appliquant à un modèle de programmation SMP où un ensemble de tâches, communicant entre elles par messages, s'exécutent en concurrence sur une plate-forme multi-cœurs avec un cache d'instruction partagé contenant plusieurs niveaux d'associativité. Les auteurs ont modélisé les différentes tâches sous la forme d'un diagramme de séquence de messages, permettant d'établir un ordre partiel sur l'exécution des tâches. Cet ordre a été utilisé pour déterminer les tâches qui, ordonnancées sur deux cœurs différents, peuvent effectivement être exécutée en parallèle, diminuant ainsi le nombre potentiel de conflits de caches qui peuvent surgir, couplé avec une optimisation de verrouillage (*Cache locking*) des blocs mémoire les plus utilisés par une tâche dans le cache L2, pour diminuer les pire temps d'exécutions calculés.

Dans [79] Lesage et al étendent leurs travaux précédents [57] en l'appliquant sur une architecture matérielle qui dispose d'une hiérarchie de caches de données partagés. La présence de tels caches en sus des caches d'instructions partagés nécessite d'adresser les problèmes supplémentaires que sont la mémoire partagée et la présence de protocoles de cohérence des caches.

Des techniques de **vérification de modèles** (*model checking*) ont également été utilisées pour modéliser le comportement des caches. Gustavsson et al [54] utilisent UPPAAL pour modéliser un système multi-cœurs avec des caches L1 privés et des caches L2 partagés et ainsi calculer le WCET de tâches exécutées. Cette méthode nécessite la modélisation de l'ensemble des instructions du système entraînant une augmentation de la taille de l'espace d'état qui rendent cette méthode utilisable uniquement pour de petits programmes.

3.1.2.2 Partitionnement

À l'inverse des approches précédentes qui s'attachent à prédire le coût des interférences inter-cœurs, les méthodes de partitionnement du cache visent à les éliminer en allouant en exclusivité une portion physique du cache à une tâche ou un cœur donné. Une distinction peut être effectuée entre les techniques *index-based* et celles *way-based* utilisées pour partager le cache, la différence entre les deux méthodes s'effectuant sur la manière dont le cache est physiquement découpé (Voir section 2.3.2).

Méthodes *index-based* Le partitionnement dit *index-based* effectue une partition du cache à la granularité d'un *set*. Il peut être mis en œuvre aussi bien en utilisant des méthodes matérielles, qui ne sont pas toujours disponibles sur les processeurs, que des méthodes logicielles.

Méthodes matérielles Le tableau 3.2 reprend la classification, effectuée par Gracioli et al [52] dans leur étude des solutions de gestion du cache, des différents travaux de partitionnement matériels *index-based* qui ont été publiées en sélectionnant ceux s'appliquant aux multi-cœurs.

Les auteurs ont effectué une classification sur l'**applicabilité** de ces méthodes en trois catégories. Les techniques qui ont été développées pour réduire les temps moyens d'exécutions des applications (**MOY**), celles qui s'appliquent aux systèmes temps réel dur (**TRD**) et celles utilisées dans les systèmes temps réel mou (**TRM**).

Les travaux proposés par Liu [84], Srikantaiah [122], Iyer [67] et Rafique [108] n'ont pas été développés pour améliorer la prédictibilité des systèmes temps réel mais pour augmenter la qualité de service et améliorer les temps d'exécutions moyens dans des système génériques. Ils peuvent néanmoins s'appliquer à des systèmes temps réel mou.

Les travaux effectués par Chousein et Mahapatra [29] et Suhendra et Mitra [125] ont, quant à eux, été développés dans l'optique de cibler des systèmes temps réel. Chousein et Mahapatra proposent une implémentation matérielle de partitionnement du cache pour augmenter la prédictibilité de systèmes multi-cœurs. Suhendra et Mitra évaluent plusieurs stratégies de partitionnement logicielles ou matérielles du cache combinées à des techniques de verrouillage de lignes de cache.

Travaux	**Applicabilité**	**Cache d'instructions ou de données**	**Technique**	**Utilisation du verrouillage**
Liu et al [84]	MOY	Deux	Modification du matériel	Non
Srikantaiah et al [122]	MOY	Deux	Modification du matériel	Non
Iyer et al [67]	MOY	Deux	Modification du matériel	Non
Rafique et al [108]	MOY	Deux	Modification du matériel	Non
Chousein and Mahapatra et al [29]	TRD	Données	Modification du matériel	Non
Suhendra and Mitra et al [125]	TRD	Données	Partitionnement logiciel ou matériel	Oui

TABLE 3.2 – Tableau comparatif des solutions de partitionnement *index-based* multi-cœurs utilisant des implémentations matérielles [52].

Méthodes logicielles La technique de *page coloring* est couramment utilisée pour effectuer un partage logiciel. Elle tire profit du système de traduction d'adresses de la *MMU* utilisé pour convertir des adresses physique en adresses virtuelles. Lorsqu'une telle opération est effectuée, l'adresse virtuelle est découpée en deux parties disjointes l'**adresse de page** composée des bits de points fort et le **décalage** au sein de la page, composé des bits de poids faibles. La partie adresse de page virtuelle est convertie en adresse de page physique en utilisant une table configurée par le système d'exploitation. La partie décalage, quant à elle, est simplement recopiée de l'adresse virtuelle vers l'adresse physique. Le système d'exploitation dispose donc d'un contrôle pour choisir à quelles zones de la mémoire physique sont associées les pages virtuelles.

Lorsqu'un cache physiquement indexé et physiquement tagué charge une ligne depuis la mémoire principale, l'adresse physique est découpée en trois parties : le **tag** composé des bits de poids forts, l'**index** au milieu de l'adresse physique, et le **décalage** composé des bits de poids faibles restants, la partie index étant utilisée pour sélectionner le *set* du cache dans lequel la donnée doit être chargée.

Lorsque la configuration de l'architecture matérielle est telle que la partie de l'adresse physique utilisée comme adresse de page par la *MMU* chevauche la partie de l'adresse physique utilisée comme index par le cache, alors le système d'exploitation dispose d'un contrôle sur les *sets* dans lesquels une page virtuelle va se trouver positionnée. Le nombre de bits qui se chevauchent entre l'adresse virtuelle et l'adresse physique détermine le nombre de « couleurs » que l'OS peut utiliser pour placer les pages. En allouant des pages d'une certaine couleur à un processus et des pages d'une autre couleur à un deuxième processus, le système d'exploitation peut s'assurer que les pages des deux processus présents dans un cache partagé ne vont jamais s'évincer.

Si la technique de coloration de page permet de déterminer dans quel *set* une donnée va être positionnée elle ne peut déterminer la voie dans laquelle elle va être placée, le choix de ce placement revenant à la politique d'allocation des données du cache.

La classification des différentes méthodes logicielles de partitionnement de type *index-based* effectuée par [52] est affichée dans le tableau 3.3.

Les approches réalisées par Tam [127], Lin [82] et Zhang [151] utilisent des techniques de coloration de pages sur des processus pour diminuer la contention spatiale sur les caches partagés d'architecture multiprocesseurs. Les travaux de Bugnion et al [21] se différencient des approches énoncées précédemment, par l'utilisation du compilateur pour récupérer des informations sur les motifs d'accès mémoire effectués par les processus d'une application parallélisée en suivant un modèle de programmation AMP (section 2.2.1). Les informations ainsi obtenues sont transmises au système d'exploitation qui utilise des techniques de coloration de page pour partitionner le cache. Toutes ces techniques ont été développées dans l'objectif d'augmenter les performances moyennes des applications.

Guan et al [53] proposent un nouvel algorithme d'ordonnancement utilisant la technique de coloration de page pour s'assurer que deux tâches utilisant des pages d'une même couleur ne puissent s'exécuter en même temps tandis que Kim et al [72] utilisent la technique de coloration de page pour éliminer les interférences inter-cœurs. Ward et al [141] proposent une nouvelle stratégie de gestion du cache qui, à la différence des travaux de Kim et de Guan s'appliquant à des tâches temps réel dur, vise des systèmes à criticités mixtes contenant un mélange de tâches temps réel dur et molles.

Travaux	Applicabilité	Cache d'instructions ou de données	OS ou compilateur	Techniques
Tam et al [127], Lin et al [82], and Zhang et al [151]	MOY	Données	OS	*page coloring*
Bugnion et al [21]	MOY	Deux	Compilateur	Compilation
Guan et al [53]	TRD	Données	OS	Ordonnancement et *page coloring*
Kim et al [72]	TRD	Données	OS	*page coloring*
Ward et al [141]	TRD/TRM	Données	OS	*page coloring*

TABLE 3.3 – Tableau comparatif des solutions de partitionnement *index-based* multi-cœurs utilisant des implémentations logicielles [52].

Méthodes *way-based* Les techniques de partitionnement *way-based* effectuent un découpage du cache à la granularité d'une *way*. Les avantages de cette technique résident, tout d'abord, dans la préservation de la structure et de l'organisation originelle des caches actuellement utilisés, l'implémentation d'une telle méthode ne nécessitant pas d'importantes modifications au niveau du matériel. De plus, cette technique permet la mise en œuvre d'un partitionnement strict pour isoler les requêtes ciblant une des partitions de celles ciblant les autres, éliminant ainsi tout problème d'interférences.

Les principales limitations de cette méthode résident dans le nombre restreint de partitions qui peuvent être utilisées ainsi que dans la granularité minimale des allocations pouvant être effectuées le tout étant déterminé par le niveau d'associativité du cache. L'augmentation du niveau d'associativité d'un cache n'est pas toujours possible car source de baisses de performances, un niveau d'associativité élevé entraînant une augmentation des temps d'accès au cache et de l'espace de stockage des *tags*.

Gracioli et al [52] ont effectués une classification des méthodes de partitionnement *way-based* en différenciant celles uniquement matérielles de celles mixtes où le matériel fournit des mécanismes de partitionnement utilisés par une couche logicielle, classification que nous avons reprise dans la table 3.4.

Les travaux de Chen et al [28] et de Sundararajan et al [126] utilisent des techniques de partitionnement pour classifier les voies du cache en deux catégories : les voies réservées aux données privées de chacun des cœurs et celle qui contiennent des zones de mémoire partagée. En effectuant cette distinction, les auteurs réduisent le taux de MISS et augmentent la qualité de service.

Muralidhara et al [93] et Qureshi et al [107] ont, quant à eux, choisi de mettre en œuvre des techniques de partitionnement dynamique. Muralidhara échantillonne les compteurs de performances pour allouer dynamiquement des parties du cache afin d'augmenter les performances du *thread* le plus lent dans le but d'augmenter les performances globales du système. Qureshi, quant à lui, propose de modifier le contrôleur du cache pour identifier les tâches dont le comportement (forte localité spatiale et temporelle, faible empreinte mémoire) les amène à

tirer profit du cache. Ils proposent ensuite d'appliquer un schéma de partitionnement du cache en favorisant ces applications.

Varadarajan et al [135] proposent un tout nouveau design de cache dit « cache moléculaire » composé d'une agrégation de molécules pouvant être associée à une application éliminant ainsi les interférences inter-cœurs.

A la différence des contributions précédentes, qui s'appliquent aux systèmes non temps réel, les travaux de Lesage et al [80] s'appliquent à des systèmes à criticités multiples contenant des tâches temps réel et non temps réel. Les auteurs ont introduit la notion de **partitions privée** composé d'un ensemble fixe de N voies pouvant être dédiées à une tâche temps réel pour contenir les blocs qu'elle a récemment accédés. Cet espace privé est dit virtuel, dans le sens ou la tâche à la garantie d'avoir l'accès en exclusivité à N blocs d'espace au sein de chaque set, sans garantie que tous ces blocs soient placés dans les mêmes voies physique. L'espace **partagé**, composé de toutes les lignes de caches qui n'appartiennent pas à une partition privée, peut être, quant à lui, être utilisé par l'ensemble des tâches accédant au cache, qu'elles aient ou non un accès à des partitions privées.

Travaux	**Applicabilité**	**Cache d'instructions où de données**	**Technique**
Chen [28]	MOY	Données	Matérielle
Muralidhara et al [93]	MOY	Données	Matérielle-logicielle
Sundararajan et al [126]	MOY	Données	Matérielle
Varadarajan et al [135]	MOY	Données	Matérielle-logicielle
Qureshi and Patt et al [107]	MOY	Données	Matérielle
Lesage et al [80]	TRD/MOY	Deux	Matérielle

TABLE 3.4 – Tableau comparatif des solutions de partitionnement *way-based* multi-cœurs n'utilisant pas de *cache-locking* [52]

3.1.2.3 Verrouillage

Les techniques de *cache locking* sont utilisées afin d'augmenter la prédictibilité des accès mémoire effectués par un programme, en verrouillant une portion des données d'une tâche dans une partie du cache de telle sorte à ce qu'elle ne puisse être évincée jusqu'à ce que le déverrouillage soit effectué. Elles reposent sur des fonctionnalités matérielles et peuvent être effectués à la granularité d'une ligne ou d'une *way* de cache.

Le tableau 3.5 reprend la classification des méthodes de *cache-locking* effectuée par Gracioli et al [52].

Asaduzzaman et al et Sarkar et al proposent deux solutions purement matérielles de verrouillage de cache. Asaduzzaman et al proposent une solution consistant à verrouiller dans le cache les blocs de mémoire qui, s'ils ne l'étaient pas, provoqueraient le plus de caches MISS.

Sarkar et al proposent un procédé utilisant du *cache locking* pour l'algorithme d'ordonnancement `PFair` afin de borner les délais de migration des tâches.

Les travaux de Suhendra and Mitra décrits précédemment dans la section 3.1.2.2 et de Mancuso et al reposent sur une combinaison de verrouillage et de partitionnement de cache, les auteurs de la deuxième solution ayant effectué une évaluation expérimentale de leurs travaux sur un matériel multi-cœur existant.

Travaux	**Applicabilité**	**Cache d'instructions ou de données**	**Technique**	**Approche logicielle**	**Statique ou dynamique**
Asaduzzaman et al [12]	TRM/TRD	Deux	Matérielle		Dynamique
Sarkar et al [114]	TRD	Deux	Matérielle		Dynamique
Suhendra and Mitra et al [125]	TRD	Deux	Matérielle-logicielle	*cache partitioning*, *cache locking*	Deux
Mancuso et al [87]	TRD	Deux	Matérielle-logicielle	*profiling*, *cache partitioning*, *cache locking*	Deux

Table 3.5 – Tableau comparatif des solutions de *cache-locking* s'appliquant aux multi-cœurs

3.1.2.4 Scratchpad

Les mémoires *scratchpad* sont similaires aux caches, dans le sens où ils sont composés, d'une petite zone mémoire placée près du processeur et pouvant être, en conséquence, accédée rapidement. Ils diffèrent de ceux-ci dans le sens où la politique de placement des données utilisées n'est pas gérée par le matériel mais logiciellement soit explicitement par le programmeur ou implicitement par le compilateur, qui va écrire des données dans la zone mémoire. Les mémoires de type *scratchpad* représentent donc une alternative viables aux caches, leur latence d'accès étant hautement prévisible, à contrario des latences d'accès aux caches qui diffèrent selon qu'un accès se traduise par un cache HIT ou par un cache MISS. En revanche, l'utilisation de telles mémoires doit être effectuée au niveau logiciel compliquant l'exécution de programmes patrimoniaux.

Un parallèle peut être établi entre l'utilisation de techniques de *cache locking* et l'utilisation de mémoires *scratchpad*, les deux techniques ayants des objectifs similaires : contrôler à tout instant quels sont les blocs de mémoires présents dans la mémoire locale. Dans [106], Puaut et Pais ont effectués une comparaison entre les techniques de *cache locking* et les mémoires *scratchpad* et ont mis en évidence deux différences majeures. Tout d'abord, l'utilisation de mémoire *scratchpad* permet d'éliminer totalement les conflits de blocs qui se produisent avec les techniques de *cache locking*, lorsque un *set* est rempli de blocs verrouillés et qu'un nouveau bloc doit être ajouté. La deuxième différence est relative à la granularité de l'allocation pouvant être réalisée avec les deux types de mémoire. Les mémoires de type caches peuvent souffrir de **pol-**

lution, les données chargées dans le cache l'étant à la granularité d'un bloc, des données ne présentant pas une forte localité spatiale ou temporelle peuvent se trouver verrouillées dans le cache. Les mémoires de type *scratchpad*, quant à elles, travaillent sur des blocs de taille variables ce qui peut générer des problèmes de **fragmentation**.

Les auteurs de PRET proposent de substituer aux caches, sources majeurs d'imprédictibilités dans la hiérarchie mémoire, des mémoires de type scratchpad [15] partitionné entre tous les *thread* matériel de telle sorte à éliminer toutes les dépendances inter-*thread*.

Les concepteurs de MERASA ont également choisi d'utiliser des mémoires *scratchpad* pour remplacer les caches, chaque *thread* critique disposant d'un accès exclusif à deux zones mémoire proches du cœur, une pour les données et l'autre pour les instructions, à vocation de caches. La zone mémoire dédiée aux instructions utilise un mécanisme matériel pour automatiquement charger la totalité du code de la fonction courante [90] tandis que l'autre contient la pile d'exécution du programme. Les *threads* non critiques disposent, quant à eux, de caches plus classiques d'instruction et de données.

3.1.3 Contrôleur mémoire

Si les solutions présentées dans l'étape précédente permettent de minimiser ou de borner la contention mémoire au niveau des caches partagés, chaque cache MISS se traduit par une requête émise vers le contrôleur mémoire pour atteindre le dernier niveau de la pyramide mémoire (Figure 2.6), à savoir, la mémoire principale.

Les caches modernes étant capable de servir en parallèle plusieurs caches MISS, le contrôleur mémoire va donc devoir traiter de multiples requêtes, provenant aussi bien des caches que des périphériques effectuant des accès *DMA*, vers les bancs mémoires en assurant le respect des contraintes temporelles nécessaires (*timing de ram*) au bon fonctionnement des puces de mémoires.

La politique d'ordonnancement choisie par le contrôleur mémoire est donc critique pour borner les temps d'exécutions des logiciels. Or nous avons vu dans la section 2.3.3.5 que les contrôleurs mémoires COTS ont tendances à prioriser les performances au détriment de la prédictibilité.

Nous avons utilisé notre classification décrite dans la section 3.1.1 pour distinguer deux approches différentes utilisées pour appliquer des propriétés de prédictibilité sur de tels composants.

La première approche emploie des techniques d'analyse de pire temps d'exécution pour borner les latences maximales pouvant pénaliser des requêtes d'accès à la mémoire. Elle peut s'appliquer aussi bien à des contrôleurs mémoires existants que sur des composants modifiés pour augmenter la prédictibilité.

Dans une deuxième partie nous allons étudier des propositions de design de nouveaux contrôleurs mémoires censés assurer, par construction, une isolation entre les requêtes mémoires issues de multiples demandeurs.

3.1.3.1 Calculs de pire temps d'exécutions

Pellizzoni et al [101] ont développé un outil de calcul de WCET pour estimer les impacts temporels générés par les interférences mémoires, sur des tâches exécutées au sein d'une architecture multi-cœurs ne disposant pas de caches partagés. Le modèle d'ordonnancement utilisé dans leur travaux, comporte des tâches périodiques partiellement préemptibles, chaque tâche étant composée d'un ensemble d'intervalles de code non préemptibles nommé *superblocks*, exécutés séquentiellement. Chaque tâche est verrouillée sur un cœur dédié, les migrations inter-cœurs étant interdites, et est ordonnancée de manière asynchrone par rapport aux tâches exécutées sur les autres cœurs. Les multiples tâches exécutées sur un même cœur, sont ordonnancées en utilisant des fenêtres temporelle de taille fixe, chaque fenêtre se voyant assigné un ensemble connu de *superblocks* et les caches privés de chaque cœur sont invalidés au début de chaque fenêtre. Les auteurs proposent une méthodologie reposant sur le calcul d'une borne maximale de la consommation mémoire d'une tâche sous la forme d'une *arrival curve*. Ils proposent d'utiliser, au choix, des méthodes expérimentales où d'analyse statique de code pour la calculer. Les auteurs vont ensuite calculer le pire temps d'exécution de toutes les tâches du système, en utilisant pour chaque tâche ordonnancée, les *arrival curves* des tâches exécutées sur des cœurs parallèles.

Wu, et al [146] ont effectué une analyse de pire temps d'exécution, sur un contrôleur dérivé des modèles utilisés sur les plates-formes COTS utilisant une politique de gestion des *row-buffer* de type *open-row* couplé avec une privatisation des bancs mémoire où chaque consommateur se voit alloué en exclusivité un sous ensemble des bancs de la mémoire physique. Les auteurs ont notamment démontré que les fonctionnalités de réordonnancent couramment utilisées dans les contrôleurs COTS peuvent conduire à des latences non bornées et proposent des modifications minimes pour augmenter la prédictibilité du contrôleur. Une analyse de pire temps d'exécution a ensuite été effectuée sur l'architecture ainsi modifiée, pour déterminer les pires latences qu'une tâche exécutée sur un cœur peut être amenée à subir lorsque des demandeurs (DMA, cœurs additionnels) effectuent de nombreux accès à la mémoire.

Kim et al ont [73] développés une modélisation boite blanche d'un contrôleur mémoire COTS utilisant une politique d'ordonnancement FR-FCFS couplé a une politique de gestion des *row-buffers* de type *open-row* pour borner les retards temporels générés par les interférences mémoires. Pour obtenir un pire temps d'exécution le moins pessimiste possible les auteurs ont utilisés deux approches orthogonales. Une approche dite *request-driven* dans laquelle ils se concentrent sur le nombre de requêtes générées par une tâche et sur les délais maximums qui peuvent les retarder. Et une approche dite, *job-driven*, dans laquelle ils étudient les ralentissements issues des interférences générées par les requêtes mémoire émises par les autres cœurs. Les deux approches ont été combinées en choisissant les résultats les plus optimistes.

Si l'approche proposée par Pellizzoni et al modélise le système mémoire comme une boîte noire, les approches développées par Wu et al et Kim et al reposent sur une approche boîte blanche dans laquelle une modélisation précise du contrôleur mémoire est effectuée. La différenciation entre les deux approches se fait sur l'hypothèse de privatisation des bancs mémoire utilisée par Wu qui limite le nombre de bancs mémoire pouvant être utilisés par une application alors que l'approche développée par Kim ne souffre pas de ces limitations.

3.1.3.2 Conception de nouveaux matériels

Akesson et al [2] ont proposé un nouveau design matériel pour concevoir un contrôleur mémoire capable de fournir des garanties, tant en termes de bande passante minimale que de latence maximale. Tout d'abord les auteurs ont défini des **groupes d'accès mémoire (*memory access groupe*)** composés d'une séquence de **commandes mémoires** pré-calculées statiquement et conçues pour ne pas interférer entre elles, par l'ajout de délais supplémentaire et l'utilisation d'une politique de gestion des *row-buffers* de type *close-row*. Les auteurs définissent au total trois différents groupes d'accès mémoire, dont la pire latence et bande passante sont connues par avance, un permettant de lire une donnée, un deuxième permettant d'écrire une donnée et un dernier utilisé pour rafraîchir la mémoire. A l'exécution le contrôleur mémoire ordonnance les groupes d'accès mémoire émis par les composants matériels consommateurs de mémoire en utilisant un algorithme à budget de priorité [3] qui permet de garantir une borne maximale sur la latence, borne qui est découplée de la bande passante allouée.

Pour augmenter la prédictibilité des accès mémoires, les créateurs de PRET [109] proposent d'utiliser une politique de gestion des *row-buffers* de type *close-row* couplé avec une politique de privatisation des bancs mémoires où chaque *thread* matériel se voit allouer en exclusivité des bancs mémoires de la RAM. Les requêtes émises par les différents consommateurs de bande passante, sont ordonnancées en utilisant une politique *round-robin* ce qui garantit l'absence de conflits entre bancs mémoires, deux commandes ciblant les même bancs ne pouvant être séquentiellement exécutées. Afin de diminuer la variabilité induite par les commandes de rafraîchissement émises périodiquement, les concepteurs de la plate-forme proposent de modifier les puces mémoires afin de pouvoir rafraîchir chaque banc mémoire séparément. Le contrôleur mémoire peut alors émettre les commandes de rafraîchissement vers les bancs mémoires utilisés par un *thread* lorsque celui-ci n'effectue pas d'accès. La combinaison des deux techniques décrites ci-dessus permet une estimation plus précise des temps d'exécutions, en évitant les confits entre les commandes de rafraîchissements et les commandes d'accès mémoire.

Paoléri et al [99] proposent un nouveau contrôleur mémoire nommé AMC pour *Analyzable Memory Controller* utilisé par MERASA et conçu pour des architectures multi-cœurs dans l'objectif de réduire le WCET des tâches temps réel. Leur contrôleur met en œuvre une politique de gestion des *row-buffers* de type *close-row* couplé à une politique *round-robin* pour servir les requêtes des différents consommateurs permettant ainsi d'établir une borne maximale, quant aux interférences générées, égale au nombre de consommateur pouvant émettre des requêtes mémoire. Le contrôleur peut être configuré pour prioriser les requêtes mémoires issues des tâches temps réel sur celle générées par les autres tâches réduisant ainsi les interférences entre les deux types d'applications. Enfin, pour réduire les interférences entre tâches, les concepteurs proposent d'isoler les requêtes émises par les différents consommateurs en instanciant une file par consommateurs.

A contrario des solutions détaillées précédemment qui reposent sur l'utilisation d'une politique de gestion des *row-buffers* de type *close-row* afin de minimiser les impacts des interférences, Krishnapillai et al [75] proposent une nouvelle approche basée sur l'utilisation d'une politique *open-row*, qui permet de tirer profit de la localité des accès mémoire, couplé à une privatisation des bancs mémoire pour supprimer les interférences entre consommateurs, cha-

cun d'entre eux se voyant attribuer en exclusivité un sous ensemble des bancs mémoires. Les auteurs ont également développé un algorithme d'ordonnancement des commandes mémoires pour diminuer l'importante latence observée lorsqu'une alternance de commandes mémoires en lecture et en écriture sont émises vers un même banc mémoire. Leur algorithme utilise la technique dite de *Rank-Switching* consistant à entrelacer les requêtes vers des *rank* différents diminuant ainsi la pire latence pouvant être générée.

3.1.4 Mémoire principale

Nous allons, dans cette partie, étudier les différentes solutions publiées pour augmenter la prédictibilité au dernier niveau de la hiérarchie mémoire à savoir la mémoire principale de type DRAM massivement utilisée dans les architectures COTS ciblées par nos travaux. Comme nous l'avons vu dans la section 2.3.3 la mémoire DRAM représente un compromis entre les coûts économiques, les limitations technologiques et les performances. Elle comporte des bus qui sont partagés entre les multiples puces et utilise des politiques d'entrelacement des données qui introduit des dépendances cachées entre des programmes exécutés sur des cœurs différents qui voient leur donnée physiquement co-localisée sur des mêmes bancs mémoire.

Les différents travaux du domaine publiés cherchent à diminuer les interférences entre des commandes mémoires issues de multiples demandeurs par des modifications logicielles ou matérielles qui trient profit de l'architecture actuelle des puces DRAM.

Dans une première partie nous étudierons une solution de coloration de bancs mémoires qui a été développée et mis en œuvre de manière purement logicielle pour, dans les deux sections suivantes, détailler des solutions logicielles de partage de canaux et d'optimisation des *row-buffer* qui nécessitent des modifications matérielles pour être concrètement mises en pratique.

3.1.4.1 Coloration de bancs mémoires

Liu et al [86], proposent une solution purement logicielle de répartition des bancs mémoires entre tâches, nommé BLPM (*Bank-level Partition Mechanism*), qui élimine les interférences inter-*threads* augmentant ainsi la prédictibilité du système. L'idée de base consiste à placer les données de chaque fil d'exécutions dans un ou plusieurs bancs mémoires dédiés pour éviter les conflits d'accès aux bancs mémoires entre des *threads* ordonnancés en parallèle.

Les auteurs proposent une technique de *bank coloring*, dans laquelle une couleur différente est attribuée à chaque banc mémoire, chaque *thread* se voyant attribuer, par le système d'exploitation qui gère la *MMU*, une ou plusieurs couleurs en exclusivité. Le nombre de couleurs disponibles dépend de l'algorithme de placement des données en RAM utilisé par le contrôleur mémoire et de la taille des pages. Une partie des bits utilisés pour colorer les bancs mémoires est également utilisé pour sélectionner l'index du cache ce qui permet, en sus, de partitionner le cache en effectuant du *cache coloring*. Les auteurs ont également proposés un algorithme logiciel pour, en l'absence de documentation, identifier les bits utilisés par le contrôleur mémoire pour effectuer les placements des données en RAM.

Une implémentation de BLPM a été effectuée au sein de l'algorithme d'allocation de blocs

mémoire du noyau Linux 2.6.32.15. L'efficacité de la solution a été évaluée en exécutant différentes combinaisons d'applications issues de la suite de test SPEC CPU2006 sur un processeur 4 cœurs avec et sans BLPM, les résultats montrant une augmentation du débit totale et de l'équité entre les différentes applications.

L'utilisation de bancs mémoire dédiés à chaque application entraîne une diminution du nombre total de bancs mémoires utilisés par chaque application. Les accès mémoires effectués par une application sont donc moins parallélisés ce qui entraîne une augmentation des conflits d'accès aux bancs mémoires inter-applications résultant en une baisse des performances. Les auteurs ont donc évalués l'impact de la réduction du nombre de bancs sur 23 benchmarks de SPEC2006 et déterminent que les applications ont besoin d'entre 8 et 16 bancs mémoire, sur les 64 bancs mémoires disponibles dans la machine, pour atteindre 90% de leurs performances.

Un autre inconvénient de la solution de coloration des bancs mémoire réside dans la granularité des allocations, qui, sur l'architecture matérielle cible, permet de distribuer des bancs mémoire à une granularité de 128 mégaoctets. Des applications peu consommatrices de mémoire se verront donc attribuer un minimum de 128 mégaoctets résultant en une sous-utilisation de la mémoire.

Yun et al [149] ont développé PALLOC qui fonctionne sur le même principe que BLPM en combinant du *bank coloring* et du *cache coloring*. Une implémentation a été effectuée sur deux plates-formes matérielles et leur solution a été évaluée sur des applications de la suite de test SPEC2006. Les auteurs ont quantifié la dégradation de performances due à la réduction du nombre de bancs mémoires et ont mesuré qu'en moyenne une application utilisant la totalité des bancs mémoires (16) subit une baisse de performances de 9% lorsqu'elle n'utilise que 4 bancs mémoire. Cette baisse de performances varie selon les applications avec une baisse maximale de 40%. Une évaluation a été effectuée pour évaluer l'impact du partitionnement du cache cumulé au partitionnement des bancs par rapport au partitionnement des bancs mémoires tout seul, les mesures montrant au final que le partitionnement du cache induit une dégradation des performances de 18%. Enfin les auteurs ont évalués l'impact de leur solution en présence de contention mémoire et ont établi qu'en l'absence de leur solution, les applications subissent un ralentissement moyen de 3.29 tandis qu'avec PALLOC le ralentissement n'est plus que de 2.13.

3.1.4.2 Partage de canaux

Muralidhara et al [94] proposent d'éliminer la contention au niveau des barrettes de RAM en répartissant les canaux d'accès à la mémoire entre les applications. En effet, chaque canal contrôlant de manière indépendante une portion distincte de la mémoire, les interférences entre applications au niveau de la DRAM peuvent être théoriquement totalement éliminées si chacune des applications utilise un canal différent pour accéder à ses données. En pratique, cette solution mise en œuvre telle quelle n'est pas utilisable, le nombre de canaux disponibles étant trop faibles pour qu'un canal puisse être dédié par application. De plus, l'utilisation d'un tel partitionnement réduit la taille de mémoire physique disponible par application et empêche l'utilisation du *channel* parallélisme ce qui se traduit par une dégradation de la bande passante mémoire utilisable par application. Les auteurs ont donc optés pour une approche intermédiaire en regroupant les applications qui interférent peu entre elles, sur un même canal.

Un algorithme de *memory channel partioning* a donc été développé qui, activé périodiquement à l'exécution, collecte le profil de consommation mémoire des applications, associe à chaque application un canal préférentiel et alloue les pages mémoire, utilisées par l'application, dans son canal préférentiel. Enfin, les auteurs évaluent leur solution comme particulièrement adaptée pour isoler des applications hautement consommatrices de bande passante mémoire mais comme inefficace pour réduire les problèmes de contention mémoire posés avec des applications peu consommatrices. D'un autre côté, ils observent que les travaux de recherches, portant sur le design de nouveaux contrôleurs mémoire, sont particulièrement efficient pour prioriser les requêtes issues d'applications peu consommatrices de bande passante. Par conséquent, ils proposent une approche hybride utilisant leur *memory channel partioning* pour solutionner le problème de la contention mémoire entre applications hautement consommatrices et utilisent un contrôleur mémoire modifié pour traiter le problème de contention mémoire avec les applications faiblement consommatrices.

Une évaluation de la solution a été effectuée sur un simulateur, les compteurs matériels nécessaires pour profiler les applications n'étant pas disponibles sur une plate-forme matérielle.

Cette solution n'est pas utilisable avec notre architecture matérielle. En effet, notre contrôleur mémoire dispose de deux canaux activables uniquement lorsque la mémoire utilisée est de type DDR2. Sur notre carte, la mémoire utilisée étant de type DDR3, un seul canal est activé ce qui rend caduc la solution proposée.

3.1.4.3 Optimisation des *row-buffer*

Sudan et al [124] ont effectués un profilage des patterns d'accès mémoire de différentes applications issues des suites de test PARSEC, SPEC, NPB, et BioBench et ont constatés que la majorité des accès mémoires sont effectués vers des zones de la taille de quelques blocs de cache présents dans un nombre restreint de pages mémoires. Afin d'optimiser les performances à l'exécution, ils proposent une approche pour co-localiser les blocs fréquemment accédés dans un même *row-buffer*.

L'identification des blocs mémoire fréquemment accédés nécessitant l'ajoute d'un trop grand nombre de compteurs dans le contrôleur mémoire (Un système avec 4 gigaoctets de mémoire et une taille de blocs de cache de 64 bytes nécessiterait 67 millions de compteurs), ils proposent donc de tracer les accès au niveau d'un regroupement de blocs nommé **micro-pages** d'une taille de 1 KiB.

Deux implémentations ont été proposées afin de co-localiser les micro-pages.

La première implémentation logicielle propose de réduire la taille des pages du système d'exploitation à une granularité d'1 KiB. Périodiquement, le système d'exploitation identifie les micros pages fréquemment accédées, en utilisant les compteurs mémoires, et les relocalise en les copiant sur un même banc mémoire et en modifiant en conséquence le mapping mémoire dans la MMU. Pour limiter l'augmentation de la taille mémoire des tables des pages provoquée par l'utilisation de pages de 1 KiB, ils proposent, en outre, d'utiliser des super pages de 4 KiB pour contenir les données peu fréquemment accédées.

Une implémentation matérielle au niveau du contrôleur mémoire a également été proposée.

Les micro-pages éligibles à une migration sont identifiées par le système d'exploitation et sont transmises au contrôleur mémoire qui les relocalise en mémoire en effectuant une copie sur un même banc mémoire. Le contrôleur mémoire contient une table interne qui permet d'effectuer la correspondance entres les adresses physiques des micros pages avant et après la recopie, évitant ainsi d'avoir à mettre à jour les tables des pages des processus.

Une évaluation de leur solution a été effectuée sur un simulateur, les matériels existants ne permettant pas de mettre en œuvre tel quelle la solution proposée, et des améliorations de performances de x11 ont été mesurées.

3.1.5 Bus matériels

Pour finir, nous allons étudier les différentes solutions proposées pour appliquer un partitionnement temporel sur les bus utilisés pour connecter l'ensemble des composants de la machine. Nous allons, tout d'abord, étudier les méthodes d'analyses de pire temps d'exécutions qui ont été développées pour borner les latences subies par des requêtes émises sur un bus utilisant une politique à temps partagé.

Nous allons, ensuite détailler de nouveaux concepts de bus matériel ayant des propriétés temps réel développé sous la forme de *network on chip*.

3.1.5.1 Calculs de pire temps d'exécutions

Wilhelm et al [143] ont démontré que la politique d'accès statique *Time division multiple access* appliquée aux bus augmente la prédictibilité facilitant ainsi la mise en œuvre de méthodes de calcul de pires temps d'exécution.

Rosen et al [110] ont effectué des calculs de pires temps d'exécutions sur des tâches exécutées sur un processeur multi-cœurs disposant de caches L1 privés et connectés à des bancs mémoires, par un bus partagé utilisant une politique TDMA, chaque processeur se voyant alloué une fenêtre temporelle pendant laquelle il a un accès exclusif au bus partagé. Les auteurs ont tout d'abord évalué, sur un exemple, l'efficacité de différentes politiques d'allocation des fenêtres temporelles, mettant ainsi en évidence l'impact de la politique du bus partagé sur les pire temps d'exécution estimés. Ils ont ensuite proposé une nouvelle méthodologie de calcul de WCET, prenant en compte les temps d'accès au bus, au sein de laquelle ils démontrent notamment que les techniques classiques de calcul de WCET appliquées au corps des boucles de code ne fonctionnent pas lorsqu'un bus partagé est utilisé. En effet, de telles techniques effectuent une estimation du nombre de cache L1 MISS générés au sein d'une boucle. Le coût d'un cache MISS étant fixe et connu par avance, il suffit donc d'ajouter au temps d'exécution de la boucle le produit du nombre de cache MISS effectués par la pénalité temporelle induite par chaque cache MISS pour obtenir le pire temps d'exécution final. Or, sur des architectures multi-cœurs utilisant un bus TDMA, le coût de chaque cache MISS est variable. Un cache MISS effectué alors que le processeur n'a pas accès au bus étant plus coûteux qu'un cache MISS qui est résolu immédiatement. Pour résoudre une telle problématique, les auteurs ont effectués un déroulage des boucles applicatives afin de calculer précisément le coût de chaque cache MISS.

Chattopadhyay et al [27] ont proposés une nouvelle technique de calcul de pire temps d'exécution qu'ils ont appliqués sur un multi-cœurs, où chaque cœur dispose de caches L1 dédiés et est connecté à un cache L2 d'instruction partagé par un bus ayant une politique statique *TDMA* avec une politique d'ordonnancement *round-robin*. Les auteurs supposent ici que les données utilisées par le processeur n'interfèrent pas avec la hiérarchie mémoire présentée ci-avant. La réalisation d'un calcul de WCET sur une architecture combinant cache et bus partagé met en exergue de nouvelles difficultés tel que l'apparition d'une dépendance circulaire entre les temps d'exécution d'une tâche et l'état du cache L2. En effet, la latence pour résoudre un cache L1 MISS dépend notamment de la présence ou de l'absence de la donnée demandée dans le cache L2. La classification d'un accès de cache L1 en L2 HIT ou MISS dépend en partie des accès effectués par les processus concurrents qui génèrent des conflits de cache. Or, l'étendue des conflits de caches résulte de la durée d'exécution pendant laquelle deux tâches s'exécutent en parallèle et la durée d'exécution des tâches découle de la durée de satisfaction des requêtes d'accès au cache. Les auteurs ont donc développés un *framework* de calcul de WCET itératif pour traiter le problème de dépendance en utilisant un analyseur de conflits de cache L2 combiné avec un outil d'analyse du bus. Pour éviter l'augmentation de complexité provoquée par la technique de déroulement de boucle proposée par [110], les auteurs ont opté pour une technique consistant à aligner chaque début de tour de boucle avec le début d'une fenêtre TDMA du bus produisant ainsi un comportement prédictible à chaque tour de boucle au prix d'une augmentation du pire temps d'exécution calculé.

Kelter, et al [69] proposent d'étendre la solution de [27] en utilisant une estimation de la position possible des fenêtres TDMA au sein des blocs de base des boucles en remplacement de l'approche par alignement pour obtenir une solution aussi précise que la solution [110] en étant aussi efficace que la solution de [27].

Chattopadhyay et al. [26] proposent un nouveau *framework* de calcul de WCET pour analyser efficacement les interactions entre des caches et bus partagés, sur des processeurs multi-cœurs utilisant des mécanismes de pipeline, et de prédiction de branchement.

3.1.5.2 Conception de nouveaux matériels

Hansson et al [50, 56] proposent une nouvelle architecture matérielle conçue pour respecter les propriété de composabilité et de prédictibilité où chaque application est exécutée sur une **plate-forme virtuelle** totalement indépendante des autres applications exécutées. Leur architecture matérielle est basée sur un ensemble de « tuiles processeurs » connectés à des « tuiles mémoires » par un *Network on chip*. L'accès au *NoC* ainsi qu'aux tuiles mémoires, partagées entre les différentes applications, est réalisée en utilisant une politique à temps partagé garantissant ainsi l'absence d'interférences. Les processeurs, conçus pour avoir une prédictibilité maximale, sont dépourvus de cache, de pipeline superscalaire et disposent d'une zone mémoire locale pour stocker les programmes à exécuter.

L'approche d'ACCROS mise en œuvre par Salloum et al. [37] a été conçue pour des applications comportant des contraintes de sûreté de fonctionnement du plus haut niveau. Les concepteurs de la plate-forme ont utilisé une approche à plus haut niveau consistant à considérer leur architecture comme un *networked multicomputer on a chip*. Les auteurs substituent

au concept de cœur la notion de **µComposant** qui, conceptuellement, représente l'équivalent d'un nœud dans un système distribué. Chaque **µComposant** a suffisamment de mémoire locale pour exécuter le code de son programme et communique avec les autres nœuds de la plate-forme en utilisant une interface de **message passing** reposant sur un *Network on chip* partagé temporellement.

3.1.6 Conclusion

Nous avons, dans cette section, effectué une étude des différentes démarches de gestion des interférences qui ont été proposées, pour chacun des composants de la hiérarchie mémoire, en utilisant une classification en trois catégories : les approches de calcul de pire temps d'exécution, celles purement logicielles et celles reposant sur la conception de nouveaux matériels.

Nous pouvons toutefois noter l'existence de transversalités entre ces multiples démarches qui, parfois, effectuent des re-conceptions de composants matériels pour appliquer des politiques logicielles afin de faciliter la mise en œuvre des méthodes de calculs de pire temps d'exécution. Alors que certaines approches matérielles proposent une re-conception totale des composants électroniques pour garantir l'isolation temporelle, d'autres se contentent de modifications minimes pour ajouter des moyens de contrôle utilisés par le logiciel pour mettre en œuvre des politiques de gestion des interférences. Si certaines des approches de calcul de WCET s'appliquent sur des composants matériels non modifiés, elles mettent généralement en œuvre des restrictions sur les modèles d'exécution logiciels pour borner les interférences et permettre le calcul de pire temps d'exécution moins pessimistes. Enfin, aussi bien les techniques logicielles que matérielles qui visent à minimiser les interférences facilitent l'utilisation des techniques de calcul de pire temps d'exécution qui voient le nombre d'interférences possibles devant être pris en compte diminuer.

Si l'on écarte les solutions matérielles, proposant une re-conception totale des cartes, les approches étudiées dans cette section ont pour inconvénients d'adresser les problèmes de contention d'un ou, tout au plus, de deux des niveaux de composants de la hiérarchie mémoire. La mise en œuvre d'approches combinées pour adresser tous les composants de la hiérarchie mémoire peut s'avérer ardue, les prérequis nécessaire à la mise en œuvre de telles solutions pouvant être antinomique. Nous allons donc étudier dans la section suivante des approches qui imposent des restrictions sur l'utilisation des ressources matérielles pour prendre en compte les interférences à un niveau global.

3.2 Approches globales

Nous allons, dans cette dernière partie, effectuer une étude des solutions qui ont été publiées afin d'ajouter des contraintes logicielles sur l'utilisation des ressources matérielles partagées pour éliminer ou réduire les interférences à un niveau acceptable. Nous reprenons la classification en deux catégories : « Contrôle des accès mémoire » et « Régulation des accès mémoire » qui a été utilisée par Girbal et al [46] dans une étude effectuant une évaluation de multiples solutions logicielles dans un contexte avionique.

3.2.1 Contrôle des accès mémoire

Nous allons, dans cette partie, étudier les solutions dites de contrôles des accès mémoire. Ces solutions visent à séquentialiser les accès concurrents à des ressources partagées éliminant ainsi, par construction, les problèmes d'interférences, deux accès concurrents à une même ressource partagée ne pouvant se produire. Les techniques classiques de calcul de pire temps d'exécutions pour processeurs mono-cœur peuvent alors être appliquées sur de tels architectures, nonobstant quelques ajustements.

3.2.1.1 Modèle d'exécution déterministe

Les modèles d'exécution de tâches déterministes visent à ordonnancer chaque accès effectué, par le matériel ou par le logiciel, à des ressources partagées, afin de supprimer ou de borner les accès simultanés qui peuvent être réalisés. Ils s'appliquent à des processeurs disposant d'un ou de plusieurs niveaux de caches privés.

En règle générale, ces modèles reposent sur un découpage des tâches en un ensemble d'intervalles exécutés séquentiellement. Une distinction est effectuée entre les **intervalles de communication**, dans lesquels le logiciel va charger et évincer des données dans un cache privé et les **intervalles d'exécution** dans lesquels la tâche utilise les données déjà présentes dans son cache sans effectuer aucun accès à la mémoire principale. En ordonnançant les phases de communication et les phases de calculs des tâches, le modèle peut supprimer ou limiter les interférences mémoire, tout en maintenant un certain niveau de parallélisme.

La figure 3.1 présente un exemple de modèle d'exécution déterministe pour une architecture 4 cœurs. Les intervalles de communication en rouge disposent de fenêtres d'accès à la mémoire tandis que les intervalles d'exécution en bleu utilisent les données présentes dans les caches. Les rectangles verts représentent les accès réels effectués à la hiérarchie mémoire dans les intervalles de communication. La politique d'ordonnancement des intervalles de communications choisie dans cet exemple vise à séquentialiser les accès à la mémoire, deux intervalles de communication sur des cœurs différents ne pouvant être exécuté en même temps.

Pellizzoni et al [100] ont présenté PREM, une réalisation de ce modèle pour des plates-formes matérielles de type COTS mono-cœur, où la mémoire principale est partagée avec des périphériques capables d'initier, de manière autonome, des transferts mémoire qui peuvent impacter les temps d'exécutions des tâches exécutées. Les auteurs proposent d'ajouter des composants électroniques de type passerelle, contenant une mémoire tampon, entre les périphériques et le bus d'accès à la mémoire. En contrôlant le *flush* des mémoires tampon, le système d'exploitation peut ainsi ordonnancer les intervalles de communication des tâches de telle sorte à ce qu'ils ne s'exécutent jamais en parallèle des accès à la mémoire effectués par les périphériques garantissant ainsi l'absence d'accès concurrents et, par là même, d'interférences. Les techniques classiques de calcul de WCET en l'absence d'interférences peuvent donc être facilement appliquées sur une telle architecture. Les auteurs ont également pris en compte la problématique d'auto-éviction, dans les phases de communication, pour éviter que le préchargement d'une ligne de cache n'évince une autre ligne chargée précédemment dans cette même phase. Pour éliminer les interférences intra-cœurs, dans lesquelles une tâche évince hors du cache les don-

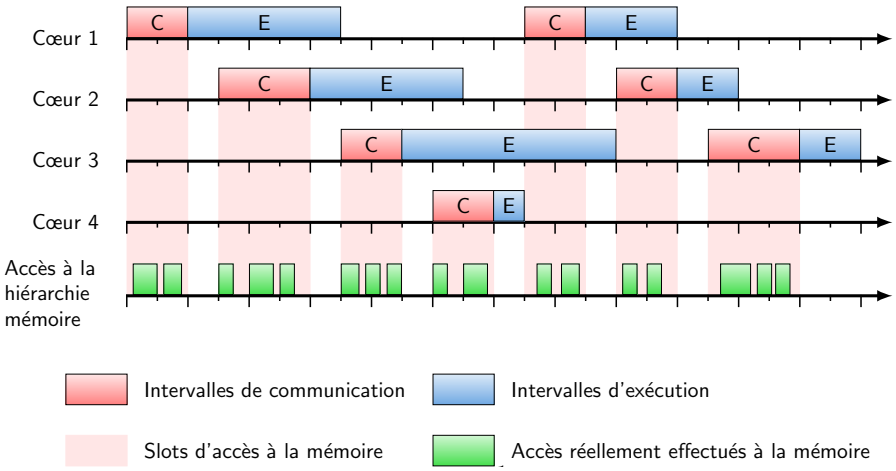


FIGURE 3.1 – Exemple de modèle d'exécution déterministe

nées d'une autre tâche, les concepteurs ont choisi de rendre les intervalles non préemptifs. Yao et al [148] ont relâché les contraintes d'ordonnancement en rendant les intervalles préemptifs mais ont partitionné le cache entre les différentes tâches.

Les concepteurs de PREM proposent, pour découper le code des tâches en de multiples intervalles, une approche manuelle s'appliquant au langage C. Le programmeur effectue une analyse des applications et de l'architecture matérielle de la plate-forme ciblée afin d'effectuer une classification des blocs du code applicatifs en deux catégories : ceux « prédictibles », annotés au sein du code source par un mot clé ad hoc, et ceux « compatibles ». Les blocs de code « prédictibles » sont, à la compilation, découpés en un intervalle de communication, ayant pour vocation de charger les données du bloc dans le cache, suivi d'un intervalle d'exécution. Ils sont donc soumis à un certain nombre de contraintes pour qu'ils ne puissent effectuer des accès mémoires dans leur phase d'exécution : pas d'accès mémoire traversant (liste chaînées), pas de récursivité, pas d'appel systèmes, d'appels à l'allocateur, d'allocations sur la pile dans des boucles, ni de préemption par le système d'exploitation. Les blocs « compatibles » sont, quant à eux, compilés sous la forme d'un unique intervalle de communication.

Plusieurs travaux ont également utilisé les modèles d'exécution déterministe sur des architectures multi-cœurs.

Yao et al [148] ont étendu les travaux réalisés dans PREM aux architectures multi-cœurs pour proposer et évaluer un algorithme d'ordonnancement nommé *Memory-centric* qui repose sur l'utilisation d'une politique d'allocation *TDMA*. Cet algorithme a été évalué par Bak et al [14] en le comparant à d'autres solutions de l'état de l'art. Alhammad et al [4] proposent un nouvel algorithme d'ordonnancement global nommé `gPREM` qui, à la différence des algorithmes précédents développés pour ordonnancer des tâches périodiques, s'applique à des tâches sporadiques à priorité fixe.

Boniol et al [19], proposent un *framework* de programmation afin d'automatiser le déploiement d'un nouveau modèle d'exécution déterministe pour processeurs multi-cœurs. Leur modèle met en œuvre des contraintes d'ordonnancement plus relâchées que celles utilisées pré-

cédemment dans le modèle dérivé de PREM. En effet, dans leur solution, les intervalles de communications exécutés sur un cœur peuvent être exécutés en parallèle de ceux exécutés sur d'autres cœurs ce qui permet d'augmenter le parallélisme du système. Leur *framework* prend en entrée les multiples tâches découpées en intervalles et un séquenceur synchrone statique qui, pour chaque cœur, décrit l'ordre, la fréquence et les échéances des intervalles des différentes tâches périodiques exécutées sur chacun des cœurs. Dans une première étape, ils évaluent le WCET de chacun des intervalles d'exécution, en utilisant des outils de calculs préexistants qui fonctionnent parfaitement sur des blocs de code non préemptifs exécutés séquentiellement en l'absence d'interférence. Les différents intervalles des tâches sont ensuite placés statiquement sur l'architecture matérielle, par un algorithme qui calcul les adresses mémoire des données et des instructions de telle sorte à ce qu'elles puissent être chargées dans le cache tout en respectant les contraintes fonctionnelles (Echéances, périodes, ordre des intervalles). Enfin, dans un troisième temps, les auteurs ont développé une méthode calcul de WCET utilisant le *Model Checker* `UPPAAL` pour évaluer les WCET des intervalles de communication en tenant compte des interférences mémoire afin de vérifier le respect des contraintes temporelles de chacune des tâches.

Durrieu et al [35] ont développé un nouveau modèle d'exécution déterministe nommé AER comportant trois phases : Acquisition, Exécution, Restitution. Dans la première phase, le code applicatif et les données utilisée dans la phase d'exécution sont chargées dans le cache, la deuxième phase s'effectuant en utilisant ces données sans effectuer aucun accès à la mémoire principale, tandis que la troisième phase effectue un *flush* des données du cache vers la mémoire principale ou vers des périphériques. Les phases d'acquisition et de restitution sont séquencées avec un ordonnancement non préemptif calculé hors-ligne de sorte à éliminer tout accès concurrents à des ressources partagées. Un cas d'utilisation mettant en œuvre ce modèle a été effectué sur un *Flight Management System*.

3.2.1.2 Ordonnancement déterministe

PikeOS est un système d'exploitation, utilisant une architecture de type micro-noyau, qui a été conçu en 2002 pour garantir l'exécution de plusieurs logiciels et systèmes d'exploitation, ayant de fortes contraintes de criticités, sur une même plate-forme matérielle. Il se présente sous la forme d'une fine couche logicielle certifiable qui ordonnance des machines virtuelles, chaque machine virtuelle exécutant une couche applicative invitée sous la forme d'un logiciel ou d'un système d'exploitation. Les couches applicatives invitées disposent de ressources matérielles dédiées, les applicatifs contenus dans une machine virtuelle peuvent donc être exécuté et certifié de manière totalement indépendante de ceux contenus dans d'autres machines virtuelles. PikeOS fournit une séparation spatiale et temporelle entre chaque applicatif, le concepteur du système peut choisir où et quand le code applicatif est exécuté. PikeOS a été certifié sur de nombreuses plates-formes mono-cœur utilisées dans des projets industriels (A400M).

Pour gérer le problème des interférences inter-cœurs, les concepteurs de PikeOS proposent [41] de donner aux concepteurs des outils permettant de définir quelles sont les couches applicatives ne devant pas souffrir d'interférences et qui, par conséquent, ne doivent pas être exécutées en parallèle d'autres applications. Ils définissent le concept de **partition de ressources**

destiné à assurer le partitionnement spatial entre les applicatifs. Une partition de ressource regroupe l'ensemble des ressources matérielles (Mémoire, périphériques externes, cœurs) requises pour l'exécution du logiciel contenu dans la partition, une même ressource matérielle pouvant être utilisée dans plusieurs partitions (Exemple : Un même cœur peut être utilisé par plusieurs partitions). Pour assurer le partitionnement temporel entre les logiciels applicatifs exécutés dans les différentes partitions, les auteurs proposent de découper le temps en une succession de **partitions temporelle**. Chaque partition temporelle se voit allouer une ou plusieurs partitions de ressource qui sont donc exécutées concurremment dans la même partition temporelle, deux partitions de ressources accédant à la même ressource matérielle ne pouvant être exécutée dans la même fenêtre temporelle.

L'exemple 3.2 illustre le fonctionnement de cette solution sur une plate-forme quad-cœur qui ordonnance 4 partitions de ressources (PR1-PR4) dans trois partitions temporelles (PT1-PT3). Le code contenu dans PR2 ne doit pas souffrir d'interférences et est donc exécuté seul. Le code contenu dans PR3 utilise trois cœurs, tandis que celui de la partition PR1 utilise deux cœurs est exécuté en parallèle de la partition R4. Les problèmes d'interférences intra-partitions (PR3 et de PR1) et inter-partitions (PR1 et PR2) doivent donc être gérées par le concepteur du système.

En utilisant cette technique, PikeOS a obtenu en 2013 le plus haut niveau de certification (SIL4 norme EN50128) pour le ferroviaire sur un processeur dual-cœur.

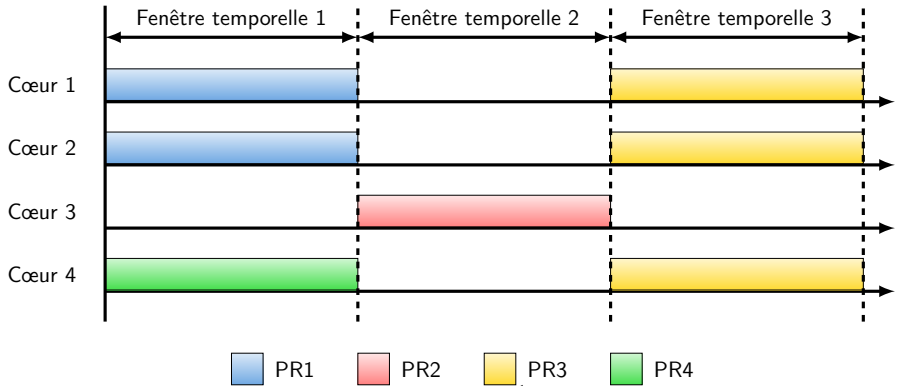


FIGURE 3.2 – PikeOS

3.2.1.3 Marthy

Jean et al [43, 47] proposent une solution, dénommée Marthy (Figure 3.3), pour éliminer les interférences inter-cœurs sans pour autant modifier le code applicatif. L'architecture matérielle cible des travaux comporte plusieurs cœurs, chaque cœur étant connecté à un ensemble de caches privés par un bus dédié. Les caches privés sont connectés au contrôleur mémoire et au contrôleur d'entrée/sortie par un bus d'interconnexion partagé. Une unité de gestion de la mémoire *MMU* est chargée d'assurer la traduction des adresses virtuelles en adresses physiques est présente et contient notamment, pour chacun des cœurs, un ensemble de *TLB* ayant pour vocation de cacher les traductions d'adresses effectuées. Le composant critique source d'inter-

férence, sur une telle architecture, est l'interconnexion qui est partagé entre tous les cœurs et contrôleurs.

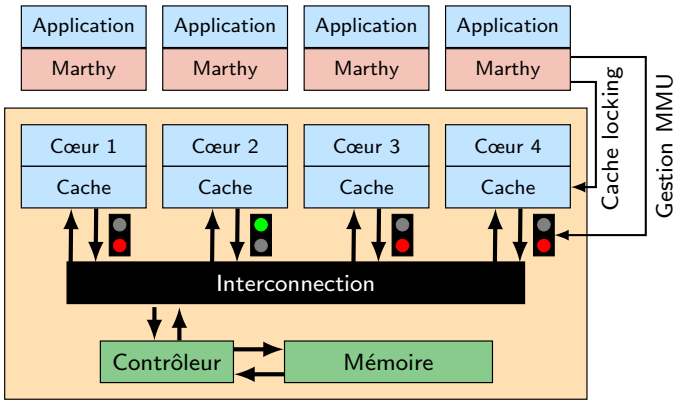


FIGURE 3.3 – Marthy

La solution proposée repose sur l'utilisation d'un système d'arbitrage TDMA pour que, durant une fenêtre temporelle donnée, un unique cœur puisse être autorisé à effectuer des accès sur le bus d'interconnexion partagé, les autres cœurs ayant le droit de continuer à s'exécuter aussi longtemps qu'ils n'effectuent pas d'accès sortant de leur cache. En séquentialisant ainsi les accès effectués aux ressources partagées, Marthy élimine les potentielles interférences fournissant un déterminisme par construction.

Les auteurs ont donc créé un mécanisme de contrôle des accès mémoires qui repose à la fois sur l'utilisation de la MMU et sur des techniques de verrouillage matériel des données dans les caches privés. Pour bien comprendre le fonctionnement de ce dispositif nous allons étudier la suite d'événements qui se produit lorsqu'un programme applicatif effectue un accès à une adresse virtuelle. Le processeur va alors regarder dans la TLB si la correspondance entre adresse virtuelle et adresse physique est bien présente et, le cas échéant, si le processus possède effectivement le droit d'accéder à une telle adresse. Si l'adresse n'est pas présente dans la TLB, le cœur va alors parcourir la table des pages du processus présente en mémoire pour résoudre la correspondance demandée, vérifier les droits du processus et charger la nouvelle correspondance dans la TLB. Lorsque le processus n'a pas les droits d'accéder à une adresse virtuelle une exception est déclenchée par le matériel sur le cœur fautif.

Au démarrage de la plate-forme, une instance du code de contrôle de Marty, destinée à assurer le bon respect de la politique de contrôle, est chargée et verrouillée dans les caches privés. L'horloge de chaque cœur est activée pour que chacune d'entre eux puisse connaître la fenêtre de temps dans laquelle il a le droit d'initier des accès mémoires. Les accès mémoires initiés par le cœur, dans une fenêtre temporelle lui donnant l'accès au bus, sont effectivement réalisés, le système de contrôle de Marthy se contenant de tracer les pages applicatives qui sont chargées et évincées hors du cache. Au moment où le cœur va rentrer dans une fenêtre temporelle ne l'autorisant pas à effectuer d'accès mémoires, le logiciel de contrôle modifie les entrées de la *TLB* pour autoriser les accès uniquement sur les pages applicatives chargées dans le cache. Si le logiciel tente alors d'effectuer un accès hors du cache, une interruption est lancée par le matériel sur le cœur fautif et est récupérée par le gestionnaire de contrôle. La tâche source

de l'interruption est alors arrêtée jusqu'à la prochaine fenêtre l'autorisant à effectuer des accès à la mémoire dans laquelle elle pourra reprendre son exécution et initier l'accès qu'elle a été empêchée de faire.

Les avantages de cette solution résident, tout d'abord, dans l'absence de modifications réalisée sur le code source des applications qui peuvent tourner tel quel ainsi que dans l'absence d'interférence inhérent à la solution. En revanche, l'efficacité de Marthy dépend de la localité des accès effectués par les tâches applicatives. Des tâches ayant une forte localité vont beaucoup utiliser leur cache et seront peu fréquemment interrompue alors que sur des tâches ayant une localité spatiale et temporelle nulle leur temps d'exécution peut être multiplié par 10. La mise en œuvre de cette solution nécessite également la présence de matériel spécifique permettant notamment de verrouiller des données dans le cache et d'écrire dans une TLB.

3.2.2 Régulation des accès mémoire

Nous allons maintenant étudier des solutions de régulation des accès mémoire qui ont pour vocation d'exécuter en parallèle des tâches concurrentes, en fournissant un dispositif pour détecter et réguler les interférences de sorte que les tâches respectent leurs contraintes temporelles.

3.2.2.1 Memguard

Yun et al [150] proposent une solution de régulation de la bande passante mémoire au niveau système, nommé « MemGuard » qui a pour objectifs de garantir une qualité de service minimale quant à la bande passante utilisables par de multiples applications tout en garantissant de bonnes performances.

Les auteurs décomposent la bande passante mémoire en deux composantes : la bande passante mémoire **minimale garantie** qui représente la bande passante minimale pouvant être délivrée en toute circonstance par le système mémoire aux processeurs. La deuxième composante, nommée bande passante mémoire **best effort**, modélise la bande passante supplémentaire pouvant être éventuellement fournie par le système mémoire. MemGuard a pour objectifs de garantir un accès par cœur à un sous ensemble de la bande passante mémoire minimale garantie tout en exploitant au maximum la bande passante *best effort*. La bande passante minimale est donc divisée en **budgets** et répartie entre les différents cœurs et un dispositif de régulation est implanté afin de stopper les sur-consommateurs.

L'architecture système de « MemGuard » (Figure 3.4) est principalement composée de deux types de composants : les **régulateurs** qui sont instanciés pour chaque cœur et le **gestionnaire de budgets** global à l'ensemble du système.

Chaque régulateur mesure et contrôle la bande passant mémoire consommée par son cœur. Le contrôle en continu des accès mémoires réalisés par les différents consommateurs étant impossible, il serait en effet nécessaire de valider chaque accès mémoire effectué par un cœur en vérifiant que le budget mémoire du cœur l'autorise, les auteurs ont donc choisi de mettre en place un mécanisme de régulation par échantillonnage. Au début de chaque période d'échan-

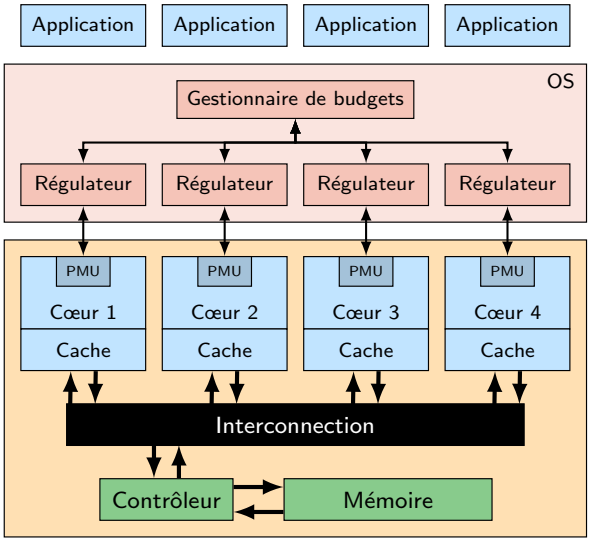


FIGURE 3.4 – Memguard

tillon, la bande passante minimale garantie est statiquement répartie entre les différents régulateurs. Le régulateur configure alors les compteurs de performance du cœur qui lui est associé de telle sorte à ce qu'une interruption soit générée lorsque le budget d'accès mémoire du cœur a été consommé. Lorsque l'interruption se produit, le régulateur stoppe toutes les tâches ordonnancées sur le cœur jusqu'à la fin de la période de régulation.

Si cette solution garantit l'isolation temporelle, chaque tâche pouvant accéder à son budget mémoire, elle entraîne de faibles performances, la bande passante *best effort* qui peut être fournie par le contrôleur mémoire n'étant pas utilisée. De plus les budgets d'accès mémoires attribués aux tâches sont alloués statiquement pour chaque tâche alors que la consommation mémoire d'une tâche évolue constamment tout au long de son exécution. Une tâche dont le budget est vide peut être stoppée sur un cœur alors que les tâches exécutées en parallèle ne consomment pas de bande passante mémoire. Pour augmenter les performances de leur solution les auteurs ont donc ajoutés deux optimisations.

Tout d'abord les auteurs considèrent que, lorsque tous les cœurs ont consommés leurs budgets, la bande passante minimale garantie a été délivrée et la contention mémoire n'est donc plus un problème. Toutes les tâches stoppées sont donc redémarrées jusqu'au début de la prochaine période de régulation. Cette optimisation permet d'utiliser la bande passante *best effort* qui peut être fournie par le contrôleur mémoire.

Afin d'optimiser la consommation mémoire de chaque tâche, les auteurs ont implantés, au sein de chaque régulateur, un oracle, chargé de prédire la bande passante mémoire que le cœur va consommer dans la prochaine période. Cet oracle prend en entrée la consommation mémoire du cœur durant la période précédente et utilise une moyenne mobile pour effectuer sa prédiction. Au début de chaque période l'oracle prédit le budget mémoire qui devrait être utilisé par le cœur. Cette prédiction est utilisée lorsqu'elle est inférieure au budget mémoire statique alloué pour le cœur, le surplus résultant étant donné au gestionnaire de budget global. Lorsqu'un cœur qui n'a pas effectué de prédiction consomme son budget il peut réclamer un

budget additionnel auprès du gestionnaire de budget global pour continuer à ordonnancer des tâches. Un cœur ayant donné une partie de son budget statique au gestionnaire global et qui voit son budget local épuisé avant la fin de la période de contrôle est victime d'une mauvaise prédiction. Dans ce cas, le cœur tente de récupérer son budget manquant auprès du gestionnaire global. Dans le cas où le budget global ne suffit pas à combler cette demande, le cœur continue son exécution jusqu'à la fin de la période de régulation en espérant qu'il puisse compenser son budget mémoire en utilisant la bande passante *best effort* éventuellement disponible. Au début de la prochaine période, le cœur victime d'une mauvaise prédiction qui n'a pas réussi à récupérer son budget mémoire, voit son budget mémoire augmenté des accès mémoires perdus pour lui permettre de compenser son retard.

Les auteurs ont évalués leur solution sur une plate-forme multi-cœurs X86 avec un processeur disposant de quatre cœurs physiques et de deux caches L2 séparés : chacun d'entre eux étant partagé entre deux cœurs. Pour mesurer le trafic mémoire généré par chaque cœur, les auteurs comptent le nombre de défaut de cache des caches L2. Comme les L2 sont partagés entre deux cœurs et que les compteurs des L2 ne sont pas capables d'identifier les cœurs consommateurs, deux des cœurs physiques ont été désactivé de telle sorte qu'un cache L2 soit utilisé par un unique cœur. Les auteurs montrent que l'optimisation de prédiction a un impact majeur sur les performances du système.

A la différence des solutions de partitionnement statique du bus, Memguard utilise une solution de régulation pour exploiter au maximum la bande passante mémoire pouvant être fournie par le matériel. L'optimisation de prédiction essentielle pour atteindre cet objectif, est adaptée aux systèmes temps réel mous où un dépassement occasionnel d'échéances ne menace pas le bon fonctionnement du système mais ne peut être utilisé dans les systèmes temps réel durs. L'architecture matérielle, utilisée pour compter le trafic mémoire généré par chacun des cœurs, n'est pas implantée dans tous les matériels.

3.2.2.2 Contrôle à l'exécution par mesure des anomalies temporelles

Kritikakou et al [76, 77] proposent un mécanisme générique pour résoudre les problèmes d'interférences temporelles dus à la contention générée par des accès concurrents effectués à des ressources matérielles partagées par des tâches exécutées en parallèle sur une architecture multi-cœurs. Leur modèle de tâches se compose d'un ensemble de tâches temps réel critiques et périodiques ordonnancées sur un unique cœur dédié, des tâches moins critiques étant exécutées sur les cœurs restants. Dans ce scénario les tâches temps réel voient leur pire temps d'exécution fortement impacté par les accès effectués aux ressources partagées au point de ne plus pouvoir respecter leurs échéances. L'exécution des tâches temps réel en isolation, c'est-à-dire en n'exécutant aucune tâche en parallèle des tâches critiques, permet le respect des échéances temporelles des tâches critiques au détriment du parallélisme, les cœurs additionnels étant utilisés par les tâches faiblement critiques uniquement lorsqu'une tache fortement critique n'est pas ordonnancée sur le système. Un troisième scénario est donc proposé pour augmenter le parallélisme, tout en garantissant le respect des contraintes temps réel, en exécutant les tâches faiblement critiques en parallèle des tâches fortement critiques aussi longtemps que les interférences provoquées par le premier type de tâches n'empêchent pas le deuxième type de tâches

de respecter leurs échéances.

Pour effectuer, à l'exécution, une surveillance du comportement temporel des tâches critiques, des points d'observations sont ajoutés dans le code binaire des tâches temps réel. Lorsqu'un point d'observation est exécuté, une condition de sûreté est vérifiée pour valider l'absence de risques de surcharge du système. Si les tâches temps réel sont trop ralenties, un processus de recouvrement est mis en œuvre par un mécanisme de contrôle. Les tâches non critiques sont alors suspendues, pour supprimer les interférences, jusqu'à la fin de la période des taches critiques. La condition de sûreté exécutée à chaque point d'observation, qui permet de déterminer la possibilité de non-respect des contraintes temporelles de la tâche temps réel, peut être modélisée par la formule mathématique suivante : $ET(x)+RWCET_{ISO}(x)+RWCET_{MAX}+t_{RT} \leq D_C$. Où $ET(x)$ représente le temps que la tâche temps réel T_C a pris pour s'exécuter jusqu'au point d'observation courant x. $RWCET_{ISO}(x)$ modélise le pire temps d'exécution en isolation depuis le point d'observation courant jusqu'à la fin de la tâche et $RWCET_{MAX}$ représente le pire temps d'exécution en contention depuis le point d'observation courant jusqu'au prochain point d'observation. Enfin, t_{RT} symbolise le temps de réaction du mécanisme de contrôle et D_C est l'échéance de la tâche temps réel T_C. La validité de la condition de sûreté émise précédemment a été prouvée par le théorème suivant : **Si le pire temps d'exécution en isolation de la tâche T_C est inférieur à l'échéance de ladite tâche alors pour toute exécution avec le mécanisme de contrôle, la tâche T_C respectera son échéance**.

Les calculs de pire temps d'exécutions requis par la condition de sûreté ($RWCET_{ISO}(x)$, $RWCET_{MAX}$) sont extrêmement coûteux en temps et ne peuvent par conséquent être totalement effectué à l'exécution par le mécanisme de contrôle. Les auteurs ont donc opté pour une approche double combinant une démarche de conception effectuée hors ligne à une démarche de contrôle effectuée en ligne. Lors de l'étape de conception un graphe de flot de contrôle est produit, depuis le binaire des applications temps réel, et est étendu par l'ajout de points de contrôle. Une grammaire a été proposée pour décrire formellement le **graphe de flot de contrôle étendu** qui sert à effectuer des pré-calculs de pire temps d'exécutions qui sont utilisés par le mécanisme de contrôle. A l'exécution le mécanisme de contrôle activé à chaque point d'observation doit recalculer la valeur de $RWCET_{ISO}(x)$ qui est modifiée à chaque point d'observation. Les auteurs ont proposés et prouvés des algorithmes, qui utilisent les données pré-calculées dans la phase de conception, pour obtenir le pire temps restant en isolation et ce sans générer d'important surcoûts temporels.

Dans [78] les auteurs ont étendus leur solution (Figure 3.5) pour prendre en compte l'exécution de plusieurs tâches critiques, exécutées en parallèle, sur plusieurs cœurs avec différentes périodes et échéances. Dans cette solution le calcul du pire temps d'exécution en isolation $RWCET_{ISO}(x)$ est modifié pour prendre en compte les interférences temporelles potentiellement générées par les tâches temps réel exécutées en parallèle. A l'exécution, chaque tâche temps réel exécute sa propre instance du mécanisme de contrôle qui vérifie si la condition de sûreté est toujours valide afin de décider si les tâches faiblement critiques doivent être suspendues en envoyant, le cas échéant, une requête à une entité centralisatrice nommé maître. Le maître, qui dispose d'une vue globale du système, est en charge de collecter les requêtes de suspension des tâches temps réel pour stopper les tâches faiblement critiques, dès lors qu'il reçoit une requête de désactivation, et les réactiver, lorsque que toutes les tâches critiques qui

ont envoyé une requête de suspension se sont terminées. Une évaluation de leur solution a également été effectuée sur une plate-forme multi-cœurs COTS de Texas Instrument [2].

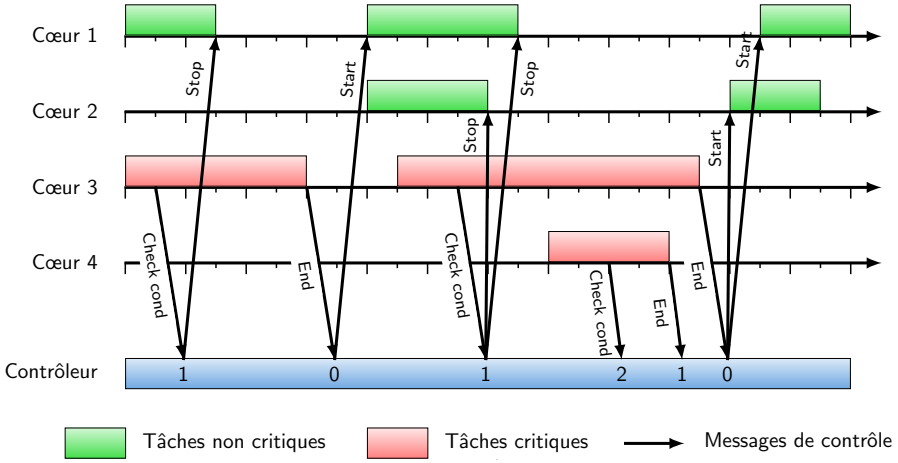


FIGURE 3.5 – Contrôle à l'exécution par mesure des anomalies temporelles

3.3 Conclusion

Nous avons, dans ce chapitre, effectué une étude des différentes approches de gestion de la contention mémoire pour des architectures multi-cœurs.

Un grand nombre des approches étudiées sont basées sur des calculs de pire temps d'exécutions. Si elles ont l'avantage d'offrir des garanties théoriques sur les temps d'exécution elles présentent certaines limites. Tout d'abord, nombre d'entre elles nécessitent une description précise de l'architecture matérielle pour être mise en œuvre, description qui n'est pas toujours fournie par les fondeurs. Ensuite, pour être efficace, ces approches requièrent une fine connaissance des logiciels exécutés sur la carte. Or, s'il est réaliste qu'une telle connaissance puisse être effective pour les applications temps réel exécutées, on ne peut présumer des applications *best effort* qui seront exécutées en parallèle.

Les approches de re-conception de matériel sont les plus séduisantes, puisqu'elles éliminent, par construction, les interférences permettant ainsi d'exécuter les logiciels sans aucunes modifications. En revanche, elles sont très coûteuses et les volumes de production des constructeurs automobiles sont trop faibles pour que nous puissions influencer les fondeurs quant aux caractéristiques implantées dans les matériels de type COTS.

Les approches globales restent attrayantes pour nos travaux. En effet, les solutions de contrôle des accès mémoires ont pour avantages d'éliminer, par construction, les interférences une seule application pouvant accéder à la hiérarchie mémoire durant une période donnée. Ce déterminisme se traduit par une dégradation des performances, la pleine et entière capacité du bus n'étant pas exploitée et peut nécessiter, au choix, une re-conception des applications (Section

2. Plateforme 8 cœurs TMS320C6678

3.2.1.1) ou la présence de fonctionnalités matérielles spécifiques permettant de contrôler les accès mémoire (Section 3.2.1.3).

Contrairement aux approches de contrôle, les approches de régulation présentent l'avantage d'utiliser plus efficacement les ressources matérielles. En effet la où les premières imposent des accès séquentiel, les secondes maintiennent autant que possible le parallélisme. La solution Memguard présentée en section 3.2.2.1 repose sur la présence de compteurs matériels permettant de mesurer le trafic mémoire généré par chacun des cœurs. Nous verrons dans la section 4.1.1.3 que de tels compteurs sont absents de notre plate-forme matérielle et que par conséquent, cette solution ne peut être utilisée telle quelle. La solution décrite dans la partie 3.2.2.2 a pour avantages de détecter les effets de la contention et non l'utilisation des ressources ce qui lui permet de détecter tout type de ralentissements pouvant se produire qu'ils soient ou non provoqué par des interférences au niveau de la hiérarchie mémoire. Elle nécessite cependant de modifier les binaires applicatifs ce qui peut rajouter un surcoût temporel aux applications temps réel et utilise des techniques de calcul de WCET.

En conclusion, malgré les nombreuses solutions de qualité proposées dans le domaine, une nouvelle approche apparaît comme nécessaire pour satisfaire aux contraintes industrielles imposées dans le cadre de ma thèse : documentation partielle limitant les approches par calculs de pire temps d'exécution, modifications de l'application temps réel interdite et carte à bas coûts sans compteurs matériels de discrimination du trafic mémoire.

Chapitre 4

Mise en évidence expérimentale du problème

Sommaire

4.1 Plate-forme matérielle . . . 76
4.1.1 Processeur . . . 77
4.1.2 Hiérarchie mémoire de niveau 1 . . . 79
4.1.3 Cache L2 . . . 80
4.1.4 Contrôleur mémoire . . . 83
4.1.5 Récapitulatif . . . 85
4.2 Stratégie de partage des ressources matérielles . . . 85
4.2.1 Partage des ressources CPU . . . 85
4.2.2 Partage des périphériques . . . 86
4.2.3 Partage de la hiérarchie mémoire . . . 87
4.3 Méthode d'évaluation . . . 88
4.3.1 Configuration du matériel . . . 89
4.3.2 Linux . . . 89
4.3.3 Benchmark : MiBench . . . 90
4.3.4 Charges . . . 92
4.4 Évaluation . . . 96
4.4.1 Coût du partage du cache L2 . . . 96
4.4.2 Impact de la contention mémoire sur les temps d'exécutions . . . 97
4.5 Conclusion . . . 97

Dans ce chapitre, nous souhaitons mettre en évidence les problèmes d'isolation temporelle entre des applications exécutées en parallèle sur une architecture embarquée multi-cœurs. Nous étudierons, dans une première partie, la plate-forme matérielle embarquée utilisée au sein de nos travaux en nous concentrant sur le système mémoire partagé entre les cœurs. Nous mettrons en évidence les impacts des choix d'implémentation matériel de cette plate-forme sur la prédictibilité des applications exécutées en parallèle.

Comme expliqué dans le chapitre 3 des méthodes de configuration fine du matériel permettent de diminuer les problèmes d'interférences inter-cœurs. Dans une deuxième partie, nous détaillerons et justifierons la stratégie de partage et de configuration que nous avons appliquée pour répartir le matériel entre les différentes applications afin de minimiser lesdites interférences.

Ensuite, dans un troisième temps, nous allons décrire la plate-forme expérimentale ainsi que les *benchmarks* que nous avons utilisés pour évaluer le problème de contention mémoire.

Enfin, dans une ultime étape, nous présenterons les résultats d'une évaluation de performances montrant que, malgré la configuration logicielle et matérielle utilisée, le problème de contention sur le système mémoire reste présent sur notre carte.

4.1 Plate-forme matérielle

Nous allons, dans cette première section, détailler la carte que nous avons utilisée pour effectuer nos travaux en portant une attention particulière sur la hiérarchie mémoire.

Ma thèse CIFRE ayant comme cadre l'équipe de R&D du constructeur Renault, le choix de la plate-forme matérielle a été guidé par les choix stratégiques de ce constructeur. Nous avons donc utilisé, comme plate-forme matérielle de référence pour nos expériences, la carte embarquée i.MX 6 SABRE Lite [11]. Cette carte de développement à bas coûts a été conçue pour exécuter des applications multimédia en utilisant les systèmes d'exploitation Linux et Android. Notons qu'il existe une variante automobile de cette plate-forme (SABRE Automotive [119]) qui a largement été utilisée par un grand nombre de fabricants et de fournisseurs du monde automobile.

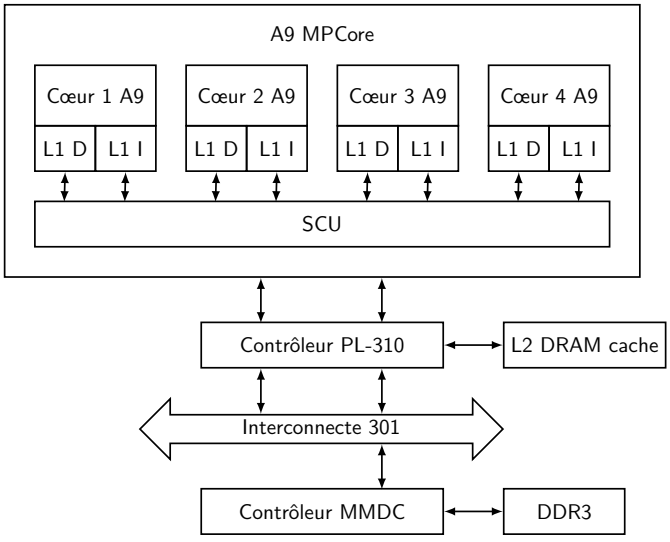


Figure 4.1 – Architecture matérielle simplifiée de la carte SABRE Lite

L'unité de calcul de la carte est composée d'un processeur **i.MX 6**, basé sur un quatre-cœurs **Cortex A9 MPCore**, connecté à un cache L2 externe **PL310** d'un mégaoctet. Le contrôleur mé-

moire **MMDC** qui gère l'accès à un gibioctet de mémoire DDR3 est connecté à un bus d'interconnexion **NIC-301** qui assure la connexion entre les consommateurs de mémoire (Processeur, GPU, ...) et différents périphériques (PCIe, MMDC, OCRAM, ...).

L'ensemble des composants sont reliés par des bus **AXI** (*AMBA eXtensible Interface*) un standard développé par ARM qui effectue une liaison point à point pour relier deux modules matériels. Un module maître initie une transaction de données en lecture ou en écriture vers un module esclave qui reçoit et répond à la transaction. L'ensemble de ces bus est équipé de deux canaux séparés qui permettent de traiter les transactions en lecture en parallèle de celles en écriture.

Nous allons maintenant, nous appuyer sur la vue d'ensemble de l'architecture de notre plate-forme, présentée dans la figure 4.1, pour détailler plus précisément les composants matériels utilisés dans nos travaux. Nous allons, dans une première partie, effectuer une description du processeur implanté au sein de notre carte pour ensuite, dans les parties suivantes, étudier les différents niveaux de hiérarchie mémoire, matérialisés par les caches L1, le cache L2 et le contrôleur mémoire, qui sont partagés entre les cœurs et les périphériques matériels.

4.1.1 Processeur

Le processeur de notre carte est composé de quatre cœurs **cortex-A9** regroupés ensemble au sein d'un unique circuit intégré intitulé **Cortex-A9 MPCore**.

4.1.1.1 Cœur cortex-A9

L'unité de calcul **Cortex-A9**, présentée dans la figure 4.2, est un processeur à haute performance et à faible consommation conçu par la société ARM qui gère l'architecture **ARMv7-A** [6] et les jeux d'instructions 32-bit ARM, 16-bit et 32-bit Thumb [10].

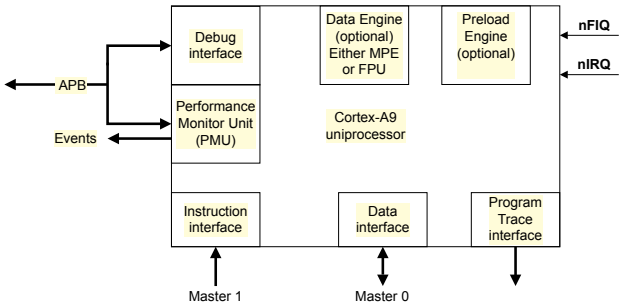


FIGURE 4.2 – Système monoprocesseur Cortex-A9 [10]

Chaque processeur possède une unité de mesure des performances (*Performance Monitoring Unit*) qui contient sept compteurs matériels pouvant être utilisés aussi bien pour récupérer des statistiques sur les opérations exécutées par le processeur (Nombre de cycles, ...) que sur les accès réalisés par le système mémoire (Cache L1 MISS, Cache L1 HIT, ...). Un des compteurs est configuré en dur pour compter le nombre de cycles effectués par le processeur tandis que les six compteurs restants peuvent être configurés pour enregistrer un des 58 événements mesurables.

4.1.1.2 Processeur cortex-A9 MPCore

Le processeur Cortex-A9 MPCore, présenté dans la figure 4.3, est constitué d'un ensemble de quatre processeurs Cortex A9 regroupés et connectés à une unité de contrôle nommée *Snoop Control unit.*

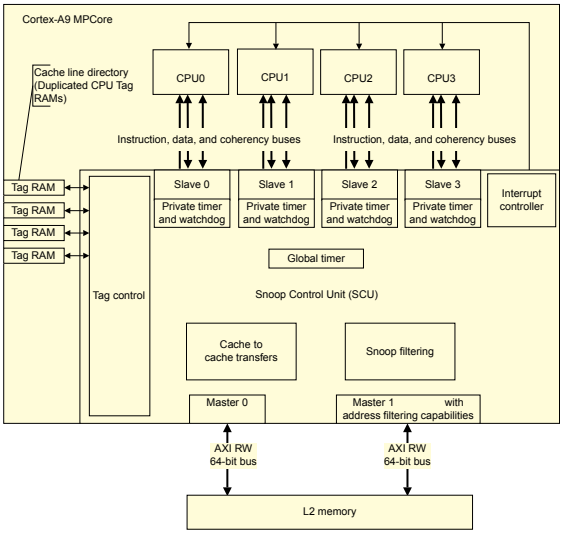


Figure 4.3 – Exemple de configuration multiprocesseur [8]

La SCU maintient la cohérence entre les caches des différents processeurs du groupe en utilisant un protocole dérivé de **MESI**. Elle arbitre également les requêtes émises par les processeurs vers les niveaux de hiérarchie mémoire supérieurs et génère les accès mémoire correspondants.

Le processeur Cortex-A9 MPCore contient, en sus, un ensemble de périphériques mappés en mémoire incluant, notamment, un temporisateur global, ainsi qu'un *watchdog* et un temporisateur privé pour chacun des processeurs Cortex A9 du groupe. Un contrôleur d'interruptions respectant l'architecture *Generic Interrupt Controller* [7] est également présent. Localisé, au sein du processeur MPCore, il a pour rôle de centraliser toutes les sources d'interruptions avant de les répartir vers les processeurs individuels.

Le processeur Cortex-A9 MPCore est connecté à un contrôleur de cache externe de type PL310 de niveau 2 par deux bus AXI 64 bits et peut, théoriquement, générer (**maître**) jusqu'à 24 transactions par processeur vers le cache L2 (**esclave**).

4.1.1.3 Impacts sur nos travaux

Si les unités de calculs cortex A9, sont indépendantes les unes des autres, le processeur MPCore représente le premier composant de notre carte qui est partagé par tous les cœurs.

La présence de la *SCU*, chargée d'assurer la cohérence entre tous les caches L1 de données, ajoute une dépendance lorsque des accès sont effectués à des données partagés. En outre, la *SCU* a également pour vocation d'arbitrer les différentes requêtes émises par les cœurs vers la hiérarchie mémoire de niveau supérieur. Or, la capacité du cache, à servir les accès mémoires

effectués étant limitée, l'ordre d'émission des requêtes vers la hiérarchie mémoire de niveau supérieur fait l'objet d'une politique d'arbitrage qui n'est pas détaillée dans la documentation matérielle dont nous disposons.

Enfin, il est important de noter que la présence de compteurs matériels, au sein de chacun des cœurs, permet d'avoir une mesure précise du trafic généré vers la hiérarchie mémoire partagée sans, pour autant, permettre de discriminer quels sont les différents niveaux de la hiérarchie qui sont traversés par les requêtes. Ce choix, effectué pour des raisons de coûts, et partagé par de nombreux fondeurs, pose l'un des défis de nos travaux. En effet, il sera impossible de déterminer quel cœur a effectué les accès mémoire mesurés.

Les sections suivantes s'attachent à décrire plus en détail les caches de premier niveau qui sont intégrés dans chacun des cœurs A9 pour ensuite porter notre attention sur le cache L2 directement connecté au processeur MPCore.

4.1.2 Hiérarchie mémoire de niveau 1

Le processeur Cortex A9 dispose de deux caches, séparément désactivables, de niveau 1, d'une taille de 32 KiB [10, 118]. Le premier cache est utilisé pour contenir les instructions de code, le cache restant étant utilisé pour charger les données. La politique de correspondance utilisée dans les deux caches est de type « partiellement associative », chaque cache étant divisé en quatre voies. La taille d'une ligne de cache étant de 32 octets soit 8 mots mémoire, chaque voie contient 256 lignes de caches. Le cache d'instructions et le cache de données sont reliés à la hiérarchie mémoire de niveau supérieur par deux bus distincts AXI de 64 bits de large, le bus *Master 0* étant utilisé par la partie de gestion des données et le bus *Master 1* par la partie gestion des instructions. La politique de gestion des écritures (*write-through* ou *write-back*) et d'allocation (*read-allocate* ou *write-allocate*), utilisée par le cache, peut être configurée par zone mémoire soit au niveau de la MMU ou au niveau de la MPU [9].

4.1.2.1 Cache d'instruction

Le cache L1 d'instructions est virtuellement indexé et physiquement tagué (*Virtually indexed, physically tagged*). Il utilise l'adresse virtuelle (*index*), émise par le processeur, pour sélectionner l'ensemble dans lequel chercher la donnée, tandis que l'adresse physique est utilisée pour déterminer si le bloc de données recherché est présent dans le cache (*tag*). L'utilisation d'une telle politique permet de diminuer la latence d'accès au cache, une ligne de cache pouvant être recherchée dans le cache en parallèle de la traduction d'adresse. Cette politique complexifie cependant la mise en œuvre de la cohérence des données partagées entre des processus exécutés sur le même cœur, une même ligne de mémoire physique pouvant être présente à deux endroits du cache. Mais, dans le cas des instructions, cette politique reste efficace. En effet, les accès sont essentiellement des lectures. La politique de remplacement du cache peut être, au choix, pseudo *round-robin* ou *pseudo-random*. Le cache L1 est connecté à une unité désactivable de prédiction du flux d'instructions du programme qui est utilisée pour précharger en avance les instructions qui vont être exécutées.

4.1.2.2 Cache de données

Le cache L1 de données est physiquement indexé et physiquement tagué (*Physically indexed, physically tagged*), ce qui augmente la latence d'accès aux données mais facilite la mise en œuvre du partage de zones mémoire entre deux processus qui utilisent le même cache, une donnée ne pouvant être présente qu'à un seul endroit du cache. La politique de remplacement des données utilisée par le cache est fixée en dur à *pseudo-random*. Le cache de données est également doté d'une unité de préchargement désactivable pour charger en avance les données qui vont potentiellement être utilisées. Un tampon *mergin store buffer* est placé entre le processeur et le cache L1 pour fusionner les écritures consécutives afin de limiter le nombre de transactions effectuées depuis le cœur vers le cache. Le cache dispose, en sus, de deux tampons de remplissage (*linefill buffer*) ce qui permet de servir deux caches MISS en parallèle. Un tampon d'éviction (*eviction buffer*) de la taille d'une ligne de cache est également présent. Il permet au processeur de propager une ligne de cache sale vers la hiérarchie mémoire de plus haut niveau sans que ledit processeur ne soit bloqué le temps de la propagation.

4.1.2.3 Impacts sur nos travaux

L'existence de caches de niveau un, privés à chacun des cœurs du processeur, entraîne, lorsque les données utilisées par le processeur sont déjà présentes dans les caches, une diminution du nombre de requêtes émises vers le système mémoire. Cette baisse, d'une part, limite le nombre d'interférences, seules les requêtes émises vers le système mémoire partagé peuvent être ralenties, et d'autre part, abaisse le nombre de requêtes envoyées vers le système mémoire ce qui se traduit par une baisse de la contention.

Les composants matériels que sont les unités de préchargement et les tampons *linefill buffer* maximisent l'utilisation du cache en préchargeant en avance les données qui vont être utilisées. Ils permettent également de découpler, le moment où les données sont requises dans le cache, du moment où elles sont réellement demandées, amortissant ainsi les effets des interférences sur le système mémoire. En effet, une requête mémoire demandée en avance par l'unité de préchargement et retardée à cause de la contention sur le système mémoire, peut arriver à temps pour être immédiatement utilisée. En revanche, l'utilisation de tels mécanismes à pour effets pervers d'entraîner un accroissement ponctuel de la demande de bande passante mémoire se traduisant par une augmentation possible des interférences.

4.1.3 Cache L2

Le cache de niveau 2, d'une taille d'un mégaoctets [118], est physiquement tagué et indexé et est partagé entre tous les cœurs (Figure 4.4). De type unifié, il peut contenir aussi bien des instructions que des données [117]. Le contrôleur de cache peut être configuré de manière logicielle de telle sorte à ce que les données présentes dans le cache L1 ne soient pas dans le cache L2 (Configuration **exclusive**) ou inversement (Configuration **inclusive**),

La politique de correspondance utilisée dans le cache est de type « partiellement associative », le cache étant divisé en seize voies d'une taille de 64 kibioctets par voie. La taille d'une

ligne de cache étant de 32 octets, soit 8 mots mémoire, 2048 lignes de caches peuvent donc être chargées dans chaque voie. La politique de remplacement du cache peut être au choix *pseudo-random* ou *round-robin*. Les politiques de gestion des écritures et d'allocations des données dans le cache sont configurables, pour chaque zone mémoire qui est chargée dans le cache, deux zones mémoire distinctes pouvant se voir attribuer deux politiques différentes. La configuration de ces politiques se fait au niveau de la MMU ou de la MPU [9].

Pour garantir un débit correct et des temps d'accès faibles à la mémoire cache, le contrôleur de cache met en œuvre une politique de *RAM banking* qui consiste à diviser la mémoire du cache L2 en quatre bancs autorisant ainsi le recouvrement des accès ce qui permet d'augmenter le débit mémoire total du cache.Le cache L2 est également doté d'une unité de préchargement désactivable capable de charger en avance des lignes provenant de la mémoire pour augmenter les performances du système.

Le cache dispose, en outre, de quatre *line fill buffers* de 256 bits partagés entre tous les ports maître, permettant de servir quatre caches MISS en parallèle, de trois *eviction buffer*, de la taille d'une ligne de cache, utilisés pour stocker les lignes évincées en attente de leur propagation vers la DRAM et de trois *store buffer* de 32 bytes capable de *bufferiser* des écritures vers la mémoire ou le cache L2 pour fusionner plusieurs transactions en écriture vers une même ligne de cache.

Le cache possède, en sus, deux *line read buffers* par port esclave : lorsqu'un cache L2 HIT se produit, les données de la ligne du cache sont tout d'abord copiées depuis le cache L2 vers un de ces tampons puis, dans un deuxième temps, transférés vers les caches L1 libérant le contrôleur de cache qui peut alors traiter d'autres accès.

Le cache L2 est aussi pourvu d'une unité de mesure des performances qui contient deux compteurs matériels configurables pouvant être utilisés pour récupérer des statistiques sur les opérations effectuées par le cache (L2 HIT, ...) sans toutefois être en mesure de discriminer le ou les cœurs sources des accès mesurés.

Le contrôleur de cache PL310 est doté d'un mécanicisme dit de verrouillage de cache (*Cache lockdown*) qui permet de contrôler le placement des données dans le cache en outrepassant la politique de remplacement.

La politique de « verrouillage par ligne » (*Lockdown by line*) permet de charger et de verrouiller des portions de données dans le cache à la granularité d'une ligne. Toutes les lignes de caches qui sont chargées, lorsque cette politique est activée, sont alors verrouillées de telle sorte à ce qu'elles ne soient jamais évincées par le contrôleur de cache. Lorsque la politique de « verrouillage par ligne » est désactivée, les lignes ultérieurement verrouillées le restent tandis que celles nouvellement allouées ne sont pas verrouillées dans le cache. Il est ultérieurement possible de déverrouiller toutes les lignes du cache qui ont été préalablement verrouillées.

La politique de « verrouillage par voie » (*Lockdown by way*) permet de verrouiller des données dans le cache à la granularité d'une voie. Elle permet d'exclure une ou plusieurs des 16 voies du cache de la politique de remplacement des données gérée par le contrôleur de telle sorte que les données présentes dans les voies exclues ne soient pas évincées.

Enfin, la politique de « verrouillage par maître » (*Lockdown by master*) dérivée de la po-

litique de « verrouillage par voie » permet à plusieurs maîtres de partager le cache L2 de la même manière que si les maîtres avaient de plus petits caches L2 qui leur étaient dédiés. Cette politique permet de décrire dans quelles voies les maîtres vont pouvoir allouer leur données, tous les maîtres ayant accès à toutes les voies pour les opérations de recherche ce qui permet de partager des données en lecture.

Le contrôleur de cache PL310 est connecté au contrôleur mémoire MMDC par deux bus AXI 64 bits connecté au bus d'interconnection NIC-301.

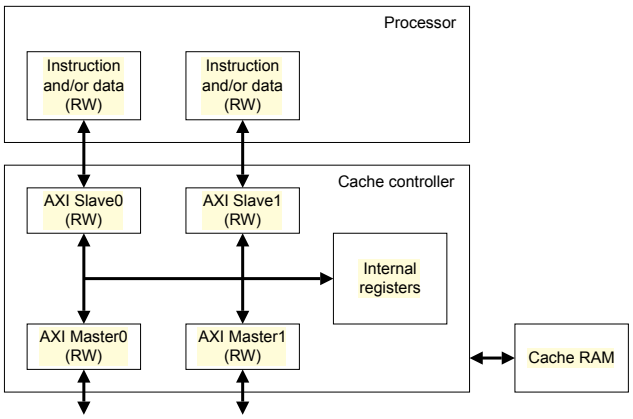


Figure 4.4 – Exemple de contrôleur de cache interfacé avec un processeur ARM [117]

4.1.3.1 Impacts sur nos travaux

Le cache de deuxième niveau, partagé entre les quatre cœurs du processeur, peut être vecteur de deux types d'interférences qui vont impacter les temps d'exécution des applications. Des **interférences spatiales** se produisent lorsqu'un applicatif, ordonnancé sur un cœur, évince les données qui ont été chargées par les autres applications exécutées en parallèle sur les cœurs restants, le comportement temporel d'une application dépendant alors des accès effectués par les autres applications. Des **interférences temporelles** peuvent également apparaître quand le cache, recevant un nombre de requêtes supérieur à celui qu'il peut traiter en parallèle, doit appliquer une politique d'arbitrage sur les requêtes à traiter.

Si la présence de compteurs matériels au sein du cache L2, permet d'obtenir une mesure globale du trafic mémoire qui est généré par l'ensemble des cœurs, elle ne permet pas de différencier les différents cœurs consommateurs. Or, nous avons vu dans la section 4.1.1.3 que les compteurs locaux à chaque cœur ne permettaient pas de mesurer la consommation effectuée dans les niveaux partagés de la hiérarchie mémoire. La mise en place d'une solution de régulation similaire à celle utilisée par MemGuard décrite précédemment dans la partie 3.2.2.1 est donc impossible sur notre plate-forme.

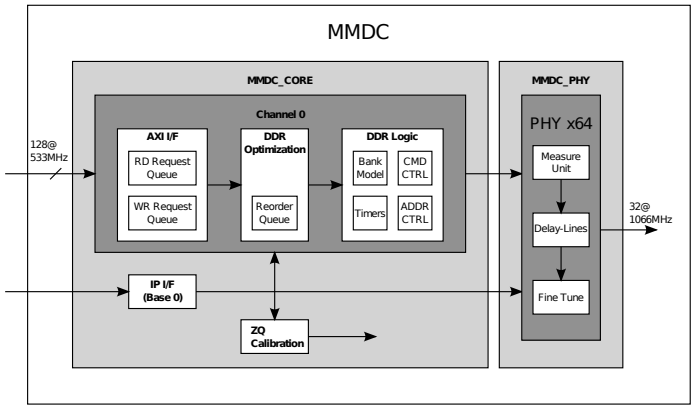


FIGURE 4.5 – Diagramme de bloc du contrôleur MMDC [118]

4.1.4 Contrôleur mémoire

Le contrôleur mémoire MMDC (*Multi Mode DDR Controller*) conçu par Freescale est chargé de gérer la mémoire DRAM de la plate-forme matérielle. Il est constitué de deux composants, le **coeur**, connecté à l'interconnecte NIC-301 par un bus AXI, gère la génération et l'optimisation des **commandes mémoire** tandis que la partie **PHY**, connectée à 1 giga de mémoire de type DDR3 [11] par un seul canal, est responsable de la gestion des *timings*.

Le cœur du contrôleur contient deux tampons *FIFO* utilisés pour stocker temporairement les requêtes d'accès mémoire émises par les consommateurs. Le premier tampon est capable de sauvegarder jusqu'à 8 requêtes d'accès en écriture tandis que le deuxième, qui dispose d'une capacité de 16 entrées, est utilisé pour sauvegarder les requêtes d'accès en lecture. Un mécanisme d'arbitrage de type *round-robin* est utilisé pour sélectionner les requêtes d'accès en lecture et en écriture qui sont en attente et les envoyer dans un tampon intermédiaire de ré-ordonnancement.

Un mécanisme d'arbitrage est utilisé pour élire une requête au sein du tampon de ré-ordonnancement et l'envoyer vers l'étage *DDR Logic*, qui le découpe en commandes mémoire transmises à la mémoire DDR à travers le composant PHY. Une fois que les accès à la mémoire sont finis, la requête élue est supprimée du tampon de ré-ordonnancement. Le contrôleur mémoire met en œuvre une politique de gestion des *row-buffer* de type *open-row* que nous avons précédemment décrite en section 2.3.3.4. Le décodage d'adresses utilise une politique de *bank interleaving* dans laquelle les lignes consécutives en mémoire sont placées dans des bancs consécutifs. Pour augmenter les performances de la mémoire, un mécanisme désactivable de prédiction permet de prédire, en parallèle du mécanisme d'arbitrage, la puce, le banc mémoire et la ligne qui vont être utilisés afin de préparer en avance la gestion des futurs accès. Le mécanisme d'arbitrage et d'élection de requête est conçu pour optimiser les accès et maximiser l'utilisation du bus DDR. Chaque requête d'accès mémoire, présente dans le tampon de ré-ordonnancement, se voit dynamiquement attribué une priorité, la requête ayant la priorité la plus forte étant sélectionnée et envoyée au composant *DDR logic*.

Le calcul de la priorité d'une requête (score) fait appel à trois facteurs différents. Le score

associé à une requête est incrémenté lorsque :

— la requête mémoire fait appel à une ligne déjà chargée dans le *row-buffer* (*row-hit*)

— le type de l'accès courant (lecture/écriture) est le même que celui précédemment effectué

— la requête n'est pas sélectionnée par l'arbitre pour éviter une famine.

L'arbitre prend également la valeur *QoS*, associée à chaque requête d'accès mémoire, pour calculer le score d'une requête. Lorsque le mode temps réel du contrôleur mémoire est activé, toutes les requêtes taguées avec un QoS maximal deviennent prioritaires sur les autres requêtes.

Le contrôleur mémoire dispose d'un dispositif de profilage permettant de récupérer des statistiques sur l'utilisation de la mémoire (Nombre d'accès effectués en lecture/écriture, trafic mémoire généré en lecture/écriture) et sur l'occupation du contrôleur mémoire (Nombre de cycles où le contrôleur est occupé). Un mécanisme de filtrage utilisant l'identifiant des bus AXI peut être mis en place pour éviter de comptabiliser le trafic mémoire généré par certains composants matériels (GPU, Ethernet, ...).

4.1.4.1 Impacts sur nos travaux

De multiples sources d'interférences sont présentes au sein du contrôleur mémoire qui va collecter l'ensemble des requêtes émises par les consommateurs pour les traduire, séquentiellement, en commandes mémoire.

Tout d'abord le nombre de requêtes pouvant être mises en attente au sein du contrôleur est borné par la taille des tampons de stockage des requêtes. Un consommateur qui émet un grand nombre de requêtes mémoire peut donc remplir les tampons du contrôleur retardant ainsi le traitement des requêtes des autres demandeurs.

Ensuite, pour maximiser les performances totales de la machine, en évitant les conflits entre requêtes, le contrôleur mémoire réordonnance les requêtes de telle sorte à maximiser le débit mémoire total de la plate-forme matérielle sans accorder une importance particulière aux performances temps réel de la machine. Un processus effectuant une suite de commandes de même type (lecture ou écriture) vers un même *row-buffer* se voyant priorisé par rapport à des processus effectuant un mixte de lectures et d'écritures. Par conséquent, un programme qui effectue des accès favorisant une bonne bande passante peut devenir prioritaire et retarder les accès effectués par un autre programme exécuté en parallèle.

L'utilisation d'une politique de *bank interleaving* permet également d'accroître les performances au détriment de la prédictibilité, les données accédées par un processus se voyant réparties sur plusieurs bancs mémoire utilisés également par d'autres consommateurs.

De plus, l'utilisation d'une politique de gestion des pages de type *open-row*, telle qu'utilisée par le contrôleur MMDC, introduit des dépendances temporelles entre les commandes mémoire. En effet, les temps d'accès à une cellule mémoire sont conditionnés par l'état du banc mémoire qui dépend des commandes exécutées précédemment. Des commandes mémoire émises par un cœur critique peuvent donc entrer en conflit avec celles émises par un cœur non critique générant ainsi des ralentissements.

4.1.5 Récapitulatif

Le tableau 4.1 contient, pour les multiples niveaux de la hiérarchie mémoire de notre carte, un récapitulatif des différentes caractéristiques matérielles qui sont partie prenante du problème de contention mémoire, à savoir, la taille et la politique de correspondance et de remplacement des différents composants. Dans une dernière colonne nous avons aussi noté les métriques disponibles pour l'élaboration d'un nouveau mécanisme de contrôle.

Hiérarchie mémoire		**Taille**	**Correspondance**	**Remplacement**	**Métriques de disponibles**
Caches L1	Instructions	32 KiB	Quatre voies	*round-robin* ou *pseudo-random*	Compteurs matériels (L1 MISS)
	Données			*pseudo-random*	
Cache L2		1 MiB	Seize voies	*round-robin* ou *pseudo-random*	Compteurs matériels (L2 MISS globaux)
Contrôleur mémoire		1 GiB	NA	NA	Compteurs matériels (Octets lus/écrits, ...)

Table 4.1 – Caractéristiques matérielles des différents niveaux de la hiérarchie mémoire.

4.2 Stratégie de partage des ressources matérielles

Nos travaux ont pour objectif de mettre en œuvre une architecture logicielle à criticité multiple dans laquelle un ensemble d'applications temps réel sont exécutées sur un cœur dédié en parallèle d'applications *best effort* exécutées sur les trois cœurs restants de notre carte. Il est donc nécessaire de partager les ressources matérielles de notre plate-forme entre les différents systèmes d'exploitation de telle sorte à ce que lesdits systèmes puissent correctement remplir leurs fonctionnalités tout en garantissant l'isolation entre les systèmes critiques et non critiques. Le but final est d'assurer le respect du principe de « composabilité », défini dans la partie 2.2.2, tout en maximisant les performances globales du système.

Nous avons fait une classification des ressources matérielles de la plate-forme devant être partagées en trois catégories : la ressource CPU, les ressources de type périphériques et les ressources de la hiérarchie mémoire.

Dans cette partie, nous allons décrire et motiver la stratégie de partage des ressources matérielles que nous avons mises en œuvre dans nos travaux.

4.2.1 Partage des ressources CPU

Pour assurer le partage de la ressource CPU, nous avions le choix entre les deux modèles décrits dans la partie 2.2.1. Avec le modèle *Asymmetric Multi Processing* un ou plusieurs cœurs

sont alloués en exclusivité à chaque système d'exploitation, les différents systèmes s'exécutant en parallèle comme s'ils étaient seuls à s'exécuter sur la plate-forme matérielle. Avec le modèle *Symmetric Multi Processing* chaque système d'exploitation s'exécute sur tous les cœurs de la plate-forme matérielle. La cohabitation entre les multiples instances des systèmes exécutés sur la plate-forme matérielle est assurée par l'hyperviseur qui va, périodiquement, exécuter chaque système.

Les systèmes temps réel, actuellement utilisés dans le milieu de l'automobile, sont majoritairement mono-cœur, la mise en production de systèmes multi-cœurs étant encore en cours de développement. En conséquence, l'utilisation d'un modèle de programmation SMP avec de tels systèmes engendre une perte de puissance de calculs, le système temps réel exécuté sur la plate-forme matérielle étant incapable de tirer parti des multiples cœurs mis à sa disposition.

Nous avons donc opté pour un modèle de programmation AMP dans lequel le système d'exploitation temps réel se voit allouer en exclusivité un cœur, les cœurs restants étant réservés au système multimédia qui est capable de tirer profit des multiples CPUs. Dans ce modèle de programmation, l'hyperviseur est chargé de garantir l'isolation entre les deux systèmes exécutés sur le matériel pour assurer que le système multimédia ne puisse perturber le bon fonctionnement du système temps réel remettant en cause le principe de **composabilité**.

4.2.2 Partage des périphériques

Le partage des périphériques entre les différents systèmes d'exploitation peut être effectué en utilisant deux stratégies.

La technique de paravirtualisation peut être utilisée pour attribuer en exclusivité certains périphériques à un système. Cette stratégie nous apparaît comme étant particulièrement appropriée pour allouer des périphériques qui sont utilisés par un seul système d'exploitation tel que le contrôleur CAN utilisé par le système temps réel ou le contrôleur Ethernet utilisé par le système multimédia. La méthode du *pass-through* qui permet de dédier des ressources matérielles à un système d'exploitation ne peut, en l'état, être utilisée sur notre plate-forme matérielle qui est dépourvue d'une IOMMU nécessaire pour assurer l'isolation spatiale entre les différents systèmes.

Certains périphériques doivent néanmoins être partagés entre l'ensemble des systèmes d'exploitation. Le GPU peut être utilisé aussi bien par le système d'exploitation temps réel pour afficher des informations véhicules à l'utilisateur que par le système multimédia pour afficher les contenus d'info-divertissement. L'utilisation de stratégies de partage sur de tels composants qui sont complexes et requièrent de bonnes performances est particulièrement difficile à mettre en place. Les solutions actuelles de partage assignent en exclusivité la ressource matérielle GPU à un des systèmes d'exploitation qui joue le rôle de serveur, les autres systèmes d'exploitation se connectant au serveur pour exporter leur affichage.

Nous avons choisi, dans le cadre de nos travaux, de ne pas investiguer les techniques de partage des périphériques matériels pour nous concentrer sur le partage de la hiérarchie mémoire qui est universellement utilisée par les programmes.

4.2.3 Partage de la hiérarchie mémoire

Les composants présents dans la hiérarchie mémoire sont les caches L1 privés à chaque cœur, le cache L2, le contrôleur mémoire, les barrettes de mémoire, le bus d'interconnexion NIC-301 et les bus AXI utilisés pour relier ensemble tous ses composants.

Nous allons aborder le partage des composants de la hiérarchie selon deux axes différents. Le partage des composants d'un point de vue spatial et le partage d'un point de vue temporel.

4.2.3.1 Isolation spatiale

L'objectif de l'isolation spatiale est de garantir que le code et les données d'une partition ne puissent être altérés par une autre partition. Sur notre plate-forme matérielle, l'hyperviseur peut utiliser l'unité de gestion de la mémoire (*Memory Management Unit*) pour partager la mémoire physique (DRAM) entre les différents systèmes d'exploitation (Partie 2.1.2.2). Chaque partition se voit assigner une zone de mémoire physique différente et la MMU garantit la séparation spatiale entre les espaces d'adressage.

Si l'isolation spatiale est nécessaire pour assurer l'intégrité physique des données utilisées par chacune des applications, elle n'est pas suffisante pour garantir que chacune des applications puisse accéder en temps voulu aux ressources qui lui sont allouées.

4.2.3.2 Isolation temporelle

L'isolation temporelle vise à assurer que les temps d'exécutions d'un système d'exploitation ne dépendent pas du fonctionnement des autres systèmes d'exploitation exécutés sur la plate-forme matérielle. Une des premières source d'interférence temporelle inter-partitions, sur notre plate-forme matérielle, réside dans les différents caches de la hiérarchie mémoire, à savoir, les caches L1 privés à chaque processeur, le cache L2 partagé entre tous les cœurs et les caches de la TLB

Nous allons maintenant détailler pour chacun des composants matériels de la hiérarchie mémoire, les différentes configurations logicielle ou matérielle, que nous avons appliquées pour renforcer l'isolation temporelle entre les commandes mémoire émises par les différents consommateurs.

Caches L1 Le modèle de programmation multi-cœurs AMP que nous avons choisi pour allouer les ressources CPU impose une allocation statique des cœurs, un cœur étant utilisé par un seul système d'exploitation. En conséquence, les caches L1 d'instruction et de données, privés à chaque cœur, **ne sont pas concernés par les interférences temporelles**, les données chargées par un système dans un cache L1 ne pouvant être évincées par celles chargées par un autre système.

Cache L2 Nous avons décidé, pour éliminer les interférences spatiales qui peuvent se produire au niveau du cache L2, de **mettre en œuvre une des techniques de partitionnement**

du cache qui ont précédemment été décrites dans la section 3.1.2.2. Nous avions donc le choix entre les techniques de partitionnement logicielles, qui reposent majoritairement sur la méthode de *page coloring*, et les techniques purement matérielles présentes sur notre plate-forme.

Nous avons fait le choix, au final, de tirer profit des modes de configuration du matériel présents sur notre plate-forme en utilisant la méthode dite de **verrouillage par maître**, décrite dans la partie 4.1.3, pour découper le cache à une granularité d'un seizième en plusieurs parties, chaque partie étant dédiée à un CPU.

Si cette technique permet d'éliminer les interférences spatiales, elle entraîne un surcoût temporel sur les applications exécutées qui disposent d'une taille de cache réduite et elle ne supprime pas toutes les interférences. En effet, la capacité du contrôleur de cache L2 à servir plusieurs cache MISS en parallèle est limitée et, même avec le cache L2 partitionné, un cœur qui génère un important nombre de MISS peut saturer les *line fill buffers* ralentissant ainsi les autres cœurs.

Mémoire principale Si une politique d'isolation temporelle peut être mise en œuvre sur les différents composants matériels que sont les cœurs, les caches L1, le cache L2, pour diminuer voir éliminer les interférences temporelles, les bus d'interconnexion AXI, l'interconnexion NIC-301, la mémoire principale et son contrôleur restent partagés.

La mémoire de type DRAM est particulièrement sensible aux interférences, des dépendances temporelles exprimées sous la forme des *timings* de RAM existant entre les différentes commandes mémoire. L'utilisation de techniques logicielles décrites antérieurement dans l'état de l'art 3.1.4 ne **peuvent malencontreusement être mises en œuvre** sur notre architecture matérielle, qui ne dispose que d'un seul canal d'accès à la mémoire et utilise une politique de *bank interleaving*.

En outre, l'algorithme de notre contrôleur mémoire a pour objectifs de maximiser les performances totales de la machine au détriment de la prédictibilité. La présence d'un mode temps réel, dans lequel les requêtes d'accès mémoire taguées avec une priorité maximale sont réordonnancées et deviennent prioritaires par rapport aux autres requêtes, paraît être la solution à ce problème. Malheureusement, le bus d'interconnexion NIC-301, utilisé pour affecter les priorités aux différentes requêtes, est uniquement capable de les affecter à la granularité d'un périphérique (CPU, GPU, IPU…). Il n'est donc **pas possible de prioriser les requêtes émises par des applications critiques** exécutées sur un ou plusieurs CPU par rapport aux requêtes émises par des cœurs qui exécutent des programmes non critiques.

Nous n'avons pas observé de mécanisme de configuration matérielle pouvant être mis en œuvre sur le contrôleur mémoire pour éliminer ou diminuer les problèmes d'interférences.

4.3 Méthode d'évaluation

Nous allons maintenant décrire la plate-forme logicielle, que nous avons déployée pour évaluer l'efficacité de ces stratégies de partage du matériel et la sensibilité de notre carte électronique face aux problèmes d'interférences temporelles.

4.3.1 Configuration du matériel

Tout d'abord, nous avons décidé d'activer l'ensemble des caches de la hiérarchie mémoire (L1, L2) ainsi que les unités de préchargement qui permettent aux applications *best effort* d'obtenir le niveau de performances nécessaire à leur bon fonctionnement. En outre, l'activation des caches L1 permet de diminuer la contention sur le système mémoire, les données chargées dans un cache privé par un cœur n'étant pas en concurrence avec celles chargées dans le cache privé d'un autre cœur.

Nous avons, pour le cache L1 d'instruction et pour le cache L2 unifié, décidé d'utiliser la politique de remplacement des données *round-robin* en place de la politique *pseudo-random* qui a pour avantage d'être plus prédictible.

Enfin, nous avons choisi d'utiliser pour les caches de niveaux un et deux de notre architecture matérielle la politique de gestion des écritures *write-back*, décrite en section 2.3.2.4, couplée à la politique d'allocation *read-write allocate* qui effectue une copie des données dans le cache aussi bien lors d'un MISS en lecture qu'en écriture. Ces politiques permettent de réduire le trafic mémoire généré vers la hiérarchie mémoire supérieure plus lente, les données étant gardées dans le cache le plus longtemps possible ce qui se traduit par une baisse de la contention sur la hiérarchie mémoire, une augmentation des performances et une baisse de la consommation électrique.

4.3.2 Linux

Comme nous ne disposions pas, pour nos travaux, du code source d'un hyperviseur porté sur notre plate-forme matérielle, nous avons donc choisi d'utiliser le système d'exploitation Linux pour créer un prototype. Notre problématique de recherche se concentrant sur les interférences temporelles inter-cœurs l'utilisation d'un système d'exploitation à la place d'un hyperviseur ne constitue pas un point bloquant.

Nous avons utilisé la version 3.0.35 du noyau Linux qui était, au début de nos travaux, la version la plus à jour officiellement supportée par le fondeur de notre matériel. La bibliothèque standard C de notre système est la version 2.20 de la bibliothèque standard GNU.

Pour empêcher les migrations inter-cœurs intempestives qui pourraient se produire, nous avons, dans nos expérimentations, verrouillé chaque processus exécuté sur un cœur dédié en utilisant le dispositif d'affinité mis à disposition par Linux, les cœurs inutilisés pendant une expérience étant désactivés.

Nous avons, afin d'émuler le comportement d'un système temps réel et de limiter les interférences entre le système d'exploitation et les applications exécutées, ordonnancé toutes les tâches en utilisant la politique `SCHED_FIFO` avec une priorité maximale. Tout processus ordonnancé avec une telle politique va, lorsqu'il est prêt à s'exécuter, préempter les processus ayant des priorités inférieures ou ordonnancés avec d'autres politiques. Il va s'exécuter aussi longtemps qu'il n'est pas préempté par un processus de plus haute priorité ou qu'il n'est pas bloqué en attente de la fin d'une opération (Entrée/Sortie, ...). L'ordonnanceur du noyau Linux contient un mécanisme de protection pour éviter qu'un processus ordonnancé avec une

politique SCHED_FIFO effectuant une boucle infinie sans exécuter d'opérations bloquantes, ne puisse empêcher, à tout jamais, les autres processus du système moins prioritaires de s'exécuter provoquant ainsi un gel du système. Ce mécanisme définit une période d'ordonnancement, au sein de laquelle un pourcentage de temps est réservé aux tâches temps réel, la période restante étant réservé aux processus exécutés avec une politique d'ordonnancement non temps réel. Nous avons donc désactivé le mécanisme de protection pour éviter les interférences qu'il induit dans les temps d'exécution des applications.

4.3.3 Benchmark : MiBench

Nous avons choisi d'utiliser, pour modéliser les applications temps réel de notre système, la suite de tests *MiBench* [55] qui a été conçue pour mettre à disposition de la communauté académique des programmes libres de droits représentatifs des applications embarquées. Cette suite de tests qui a été citée plus de 2700 fois [1] est, de fait, une référence dans le domaine académique. Elle contient 35 applications embarquées majoritairement écrites en C et fournies, lorsque c'est pertinent, avec deux jeux de données. Le petit jeu de données représente une utilisation légère du programme alors que le jeu de données étendu est plus intensif et représentatif des comportements observés dans le monde réel.

Le domaine de l'embarqué se caractérise par une importante hétérogénéité aussi bien dans les architectures matérielles employées que dans les logiciels utilisés. L'éventail des solutions, mises en œuvre dans le domaine de l'embarqué, recouvre une large diversité allant d'applications exécutées en *bare metal* sur de petits microcontrôleurs à des systèmes d'exploitation déployés sur des téléphones dont les fonctionnalités se rapprochent de celles d'un ordinateur. La création d'une unique suite de test ne permettant pas de prendre en compte cette diversité, les créateurs de MiBench, ont donc opté pour une approche plurielle dans laquelle six catégories de tests sont créées, chaque catégorie étant représentative d'une partie du marché applicatif de l'embarqué.

La catégorie *automobile et contrôle industriel* a, pour objectif, de caractériser l'utilisation des processeurs embarqués dans les systèmes de contrôle tel que les contrôleurs d'airbag, les contrôleurs moteurs et les systèmes effectuant des acquisitions depuis des capteurs. Ces systèmes ont besoin de bonnes performances dans les calculs mathématiques, les opérations effectuées sur les bits, les opérations d'entrée/sortie et les opérations de recherche. La catégorie *réseau* modélise les applications embarquées dans les équipements réseaux tel que les routeurs et commutateurs. Le travail effectué par ce type de programme inclut des algorithmes de calculs de plus court chemin, des recherches dans les arbres et les tableaux et des opérations d'entrée/sortie. La catégorie *sécurité* contient principalement des algorithmes de chiffrement, de déchiffrement et de hachage. La catégorie *appareils électroniques grand public* a pour objectifs de représenter les appareils électroniques tels que les scanners, les caméras et appareils photo numériques. La suite de test de cette catégorie se concentre principalement sur les applications multimédia tel que le décodage de musique, le traitement d'image. La catégorie *bureautique* contient des algorithmes de manipulation de texte utilisés dans les applications d'impression, de fax et de traitement de texte. Enfin, la catégorie *télécommunications*, contient

1. Google Scholar, 20 janvier, 2016

des algorithmes de traitement du signal, d'encodage et de décodage, d'analyse de fréquence et de somme de contrôle utilisés dans les logiciels de communications sans fil.

Nous avons effectué une sélection sur les logiciels que nous avons utilisé dans nos tests. Nous avons exclu toutes les applications de la suite appareils électroniques grand public et de la suite bureautique, ainsi que l'application `gsm` de la suite télécommunication, qui ne sont pas représentatives d'applications temps réel. Nous avons également exclus les programmes `blowfish` et `pgp` de la suite sécurité qui sont partiellement écrits en assembleur X86 et ne sont donc pas portables sur notre architecture. Le tableau 4.2 contient les applications sélectionnées pour modéliser les tâches temps réel de notre système.

Catégorie	**Application**	**Description**
Automobile	basicmath	Opérations mathématiques basiques, généralement exécutées par le logiciel (Conversion de degrés en radian, Racine carré, ...), utilisées, par exemple, pour calculer la vitesse d'un véhicule.
	bitcount	Algorithmes développés pour mesurer la capacité du processeur à effectuer des opérations de calcul du nombre de bits présents dans un tableau d'entiers.
	qsort	Tri d'un tableau de chaines en utilisant l'algorithme de tri rapide *quicksort*
	susan	Logiciel de détection d'image initialement développé pour reconnaître les angles et les arêtes d'images du cerveau issues d'un IRM
Réseau	dijkstra	Recherche de plus court chemin dans un graphe implémenté sous la forme d'une matrice adjacente en utilisant l'algorithme de dijkstra
	patricia	Structure de données compacte obtenue à partir d'un arbre préfixe en fusionnant chaque nœud n'ayant qu'un seul fils avec celui-ci permettant ainsi de réduire les temps de parcours de la structure de donnée au dépend d'une augmentation de la complexité du code.
Télécommunications	adpcm	Algorithme de compression de données avec perte
	CRC32	Algorithme de calcul d'un contrôle de redondance cyclique 32 bit effectué sur le contenu d'un fichier.
	FFT/ FFTI	Effectue une transformée de Fourier et son inverse sur un tableau de données. La transformée de Fourier est utilisé dans le domaine de traitement du signal pour identifier les fréquences contenues dans un signal d'entrée.
Sécurité	rijndael	Algorithme de chiffrement de données.
	sha	Algorithme de hachage qui produit un message d'une taille de 160 bit pour une entrée donnée.

TABLE 4.2 – Applications sélectionnées

Les programmes ont été compilés avec le compilateur croisé GGC 4.9.1 et avec les options `-O2`, `-march=armv7-a`, `-mthumb-interwork`, `-mfloat-abi=hard`, `-mfpu=neon`, `-mtune=cortex-a9`

qui permettent d'optimiser les programmes compilés pour notre plate-forme matérielle.

Les différents jeux de données utilisés par les applications de la suite MiBench sont fournis sous la forme de fichiers texte ou binaire. Ils contiennent des données qui, dans les plates-formes embarquées existantes, sont habituellement fournies par des périphériques externes tels que des caméras embarquées, des contrôleurs réseaux ou des microphones qui interagissent avec la mémoire passant par le *Direct Memory Acces*. Les fichiers de données d'entrée des programmes et les sorties effectuées par ces mêmes applications sont réalisées en utilisant les fonctions d'entrée/sortie fournies par bibliothèque standard du C. Nous avons donc, pour obtenir un comportement le plus réaliste possible, placé les données applicatives dans un système de fichier en mémoire principale (*TMPFS*).

Le tableau 4.3 contient, pour chaque application sélectionnée dans la suite de test, le temps d'exécutions maximal, moyen et l'écart type. Les applications ont été exécutées en isolation sur le cœur 0 et le cache L2 n'a pas été partitionné. Chaque expérience a été exécutée 150 fois et les 20 premières exécutions ont été supprimées pour minimiser la variabilité de l'expérience.

4.3.4 Charges

Pour évaluer le degré de sensibilité de notre plate-forme matérielle au phénomène de contention mémoire, nous avons opté pour une approche expérimentale. Nous avons donc développé un *microbenchmark* spécifique, dont le code source écrit en assembleur, présenté dans la figure 4.6, est chargé de stresser la hiérarchie mémoire de notre plate-forme matérielle en générant des requêtes d'accès mémoire.

Notre programme charge comporte deux boucles imbriquées.

1. La boucle interne nommée `w_stress_loop` itère sur un tableau et effectue des accès en écritures contigus de la taille d'un mot mémoire pour un total de 32 octets de données écrit à chaque tour de boucle, soit la taille d'une ligne de cache. Elle utilise, pour effectuer les écritures, l'instruction `stmgeia` qui, simultanément, génère un accès à la mémoire et incrémente l'itérateur sur le tableau de données permettant ainsi d'obtenir un « meilleur » débit mémoire.
2. La boucle externe, `w_outer_loop` permet de contrôler le temps d'exécution de la charge en répétant le parcours de tableau autant de fois que nécessaire.

L'objectif principal, qui a guidé le développement de notre programme charge, était de générer un maximum d'interférences sur le système mémoire pour ralentir au plus les programmes exécutés en parallèle sur les autres cœurs de la machine.

Pour atteindre cet objectif, nous souhaitions solliciter l'ensemble de la hiérarchie mémoire partagée de la plate-forme matérielle allant du cache L2, en passant par le contrôleur mémoire, jusqu'aux puces de RAM. Or, notre architecture dispose de deux niveaux de caches entre les cœurs, source des requêtes de consommation mémoire, et le dernier niveau de la hiérarchie mémoire. Il était donc nécessaire que les accès mémoire initiés par le programme sortent au maximum des caches pour atteindre le dernier niveau de la hiérarchie mémoire.

Nous souhaitions également maximiser la bande passante mémoire pour, par un grand nombre de requêtes, saturer les bus d'interconnexion et le contrôleur de RAM.

Application		Description	Temps moyen d'exécution (ms)	Temps maximal d'exécution (ms)
basicmath	large	auto : math calculations	54.82 ± 0.03	54.94
	small		12.31 ± 0.01	12.33
bitcount	large	auto : bit manipulation	413.63 ± 14.50	449.63
	small		27.46 ± 1.12	30.52
qsort	large	auto : quick sort	23.44 ± 0.08	23.59
	small		18.28 ± 0.05	18.38
susan -e	large	auto : image recognition	56.54 ± 0.08	56.75
	small		2.08 ± 0.02	2.13
susan -s	large	auto : image recognition	270.97 ± 0.01	270.99
	small		17.96 ± 0.01	17.98
susan -c	large	auto : image recognition	23.80 ± 0.05	23.92
	small		1.08 ± 0.02	1.15
adpcm	large	telecom : speech processing	550.29 ± 0.08	550.49
encode	small		30.83 ± 0.01	30.88
adpcm	large	telecom : speech processing	523.33 ± 0.07	523.52
decode	small		26.06 ± 0.01	26.09
fft	large	telecom : FFT	120.17 ± 0.21	121.00
	small		8.40 ± 0.04	8.48
fft -i	large	telecom : inverse FFT	122.30 ± 0.17	122.98
	small		17.89 ± 0.05	18.01
crc32	large	telecom : cyclic redundancy check	3068.97 ± 0.06	3069.18
	small		157.59 ± 0.01	157.62
patricia	large	network : tree structure	283.28 ± 2.97	289.77
	small		49.42 ± 0.06	49.58
dijkstra	large	network : shortest path	228.89 ± 0.21	229.33
	small		53.06 ± 0.03	53.14
sha	large	security : secure hash	82.20 ± 0.02	82.26
	small		7.63 ± 0.01	7.65
rijndael	large	security : block cipher	285.96 ± 0.30	286.92
encode	small		27.02 ± 0.11	27.25
rijndael	large	security : block cipher	264.83 ± 0.15	265.32
decode	small		24.89 ± 0.07	25.09

TABLE 4.3 – Temps d'exécution des applications MiBench exécutées avec le cache non partitionné

Enfin, nous avons également essayé de tirer profit de la politique d'ordonnancement des requêtes du contrôleur mémoire pour prioriser nos requêtes par rapport à celles issues des autres consommateurs.

Nous allons, dans la première sous partie de cette section, détailler, par une analyse du comportement de notre application, les éléments de conceptions qui nous ont permis de stresser le dernier niveau de la hiérarchie. Ensuite, dans un second temps, nous allons étudier le débit mémoire maximal atteignable par notre application, pour, dans une troisième partie, aborder

les impacts d'un tel trafic sur la politique d'arbitrage du contrôleur mémoire.

4.3.4.1 Stress de l'ensemble de la hiérarchie mémoire

Nous allons maintenant effectuer une analyse du comportement de notre application « charge » exécutée sur le système en l'absence d'autres applications avec les caches L1 et L2 vides ou remplis de données invalides au début de l'exécution du programme.

La politique de remplacement des données utilisée par le cache L1 pour déterminer dans quelle voie du cache un bloc de donnée chargé depuis le niveau de la hiérarchie mémoire supérieure doit être placé est de type *pseudo-random*. Cette stratégie, vise tout d'abord, à remplir les voies du cache qui sont libres. Quand toutes les voies contiennent des données valides, une voie est tirée aléatoirement parmi celles disponibles. Lors du premier tour de la boucle `w_outer_loop`, la boucle `w_stress_loop`, qui effectue des accès contigus en écriture, va, en parcourant les 32768 premiers octets du tableau en mémoire remplir, ligne après ligne, la totalité du cache L1 générant 1024 cache MISS.

La politique de gestion des écritures utilisée pour le cache L1 étant de type *Write-Back Read-Write-Allocate* chaque cache MISS entraîne un chargement de la ligne de cache contenant la donnée depuis les niveaux de la hiérarchie mémoire supérieure vers le cache L1. Lorsque les 32768 premiers octets du tableau ont été écrits en mémoire, le cache L1 est rempli de données sales. Tout nouveau cache MISS effectué dans le cache nécessite d'évincer une ligne de cache qui contient des données qui n'ont pas encore été propagées aux niveaux de la hiérarchie mémoire supérieure avant de charger les nouvelles données voulues.

Le même comportement est observé au niveau du cache L2 qui est inclusif au cache L1 et qui utilise la politique de remplacement des données *round-robin* avec une politique de gestion des écritures utilisée de type *Write-Back Read-Write-Allocate*. Lorsque le cache est vide, le premier mébioctet de données parcouru par la boucle `w_stress_loop` va générer 32768 cache MISS et autant d'accès mémoire pour charger les blocs de données voulu depuis la mémoire vers le cache. Les 32768 caches MISS générés par le parcours du mégaoctet restant vont entraîner des accès en écriture pour écrire les lignes de cache en mémoire principale suivi d'accès en lectures pour charger les blocs de données voulus dans le cache.

La taille du tableau parcouru par la boucle `w_stress_loop` est d'une taille de deux fois la taille du cache L2 soit 2 mébioctets. Cette taille permet de s'assurer que les données chargées par la boucle `w_stress_loop` ne tiennent pas dans le cache L2, évitant ainsi qu'un bloc de données chargé par la boucle `w_stress_loop` lors de l'itération i de la boucle externe `w_outer_loop` ne soient encore présent lors de la $i + 1$ itération diminuant ainsi le débit mémoire généré par la charge.

4.3.4.2 Maximisation de la bande passante mémoire

L'utilisation d'écritures pour solliciter le bus mémoire couplé à une politique de gestion des écritures de type *write-back* et à une politique d'allocation de type *write-allocate* a, pour avantage, de générer un plus fort trafic mémoire qu'une charge qui n'utiliserait que des lectures.

En effet, lorsque un mébioctet de donnée a été parcouru et que le cache est rempli de données sales, chaque écriture effectuée génère deux accès mémoire, un accès en écriture pour écrire la ligne de cache sale et un en lecture pour charger la ligne de cache contenant la donnée demandée dans le cache. Ces accès supplémentaires se traduisent par une bande passante générée plus importante, la hiérarchie mémoire étant dotée de *line fill buffers* et d'*eviction buffer* qui permettent de servir plusieurs accès mémoire en parallèle.

Nous avons, en utilisant les compteurs mémoire du contrôleur MMDC, mesuré la bande passante de ce programme exécuté en isolation sur la carte qui atteint 2020 MB/s.

4.3.4.3 Politique d'ordonnancement des requêtes du contrôleur mémoire

L'utilisation d'accès séquentiel permet également de générer le maximum de contention mémoire, le contrôleur MMDC ayant une politique visant à favoriser les requêtes du même type et faisant des accès contigus.

```
1  .arm
2  .text
3  .align
4
5  .global stress_write_no_delay
6  stress_write_no_delay:
7      push {r4-r12,lr}        @ Sauvegarde des registres sur la pile
8
9      ldmia r0, {r0-r2}       @ r0 -> Adresse du tableau
10                             @ r1 -> Taille du tableau
11                             @ r2 -> Nombre d'iteration sur le tableau
12
13     mov r11, #0
14     mov r12, #0
15
16 w_outer_loop:
17     mov r6, r0              @ r6 <- Iterateur du tableau
18     mov r7, r1              @ r7 <- Taille restante du tableau
19
20 w_stress_loop:
21
22     stmgeia r6!, {r11,r12}  @ Ecrit a l'adresse du tableau contenu dans r6 le contenu
23                             @ des registres r11 et r12 et incremente l'iterateur r6
24     stmgeia r6!, {r11,r12}
25     stmgeia r6!, {r11,r12}
26     stmgeia r6!, {r11,r12}  @ A ecrit 32 octets soit une ligne de cache L2
27
28     subs r7, #32            @ Taille du tableau -= 32 bytes
29     bgt w_stress_loop       @ Si taille restante du tableau > 0, rebouclage
30
31     subs r2, #1             @ Decremente le nombre d'iterations
32     bgt w_outer_loop        @ Si nombre d'iteration sur le tableau > 0, rebouclage
33
34     pop {r4-r12,pc}
```

FIGURE 4.6 – Boucle principale du programme « charge » développé pour solliciter la hiérarchie mémoire

4.4 Évaluation

Dans cette partie nous allons, par une approche expérimentale, évaluer la pertinence de nos stratégies de partage et d'allocation des ressources matérielles de notre carte. Dans un premier temps, nous allons analyser l'impact du partitionnement du cache sur les performances des applications de MiBench exécutées en isolation pour, ensuite, observer dans quelle mesure le partitionnement du cache permet de résoudre le problème d'interférences inter-applications.

4.4.1 Coût du partage du cache L2

Le partage du cache L2 entraîne une réduction de la taille totale de cache disponible par application ce qui peut impacter les performances des programmes fortement consommateur de mémoire.

La figure 4.7 montre l'impact de la politique de partitionnement du cache sur les performances des applications de MiBench exécutées seules. Le surcoût est calculé par rapport aux applications MiBench exécutées sans partitionnement. Chaque application a été lancée 150 fois et nous avons supprimé les 20 premières exécutions. Sur les 130 exécutions restantes, nous avons, dans une approche pire cas, utilisé le temps maximal d'exécution aussi bien comme temps de base applicatif que pour le calcul du surcoût temporel. Nous avons étudié deux configurations de partage du cache. Dans la première, le cache est divisé en deux, une partie étant réservée au cœur 0, qui exécute l'application MiBench, le reste étant alloué aux trois cœurs restants. La deuxième configuration choisie est asymétrique, 1/4 du cache étant dédié au cœur 0, les 3/4 restants étant utilisés par les trois autres cœurs. Dans cette configuration, les *threads* des applications charges qui modélisent des programmes applicatifs *best effort* peuvent utiliser une plus grande partie du cache L2 ce qui augmente leurs performances potentielles.

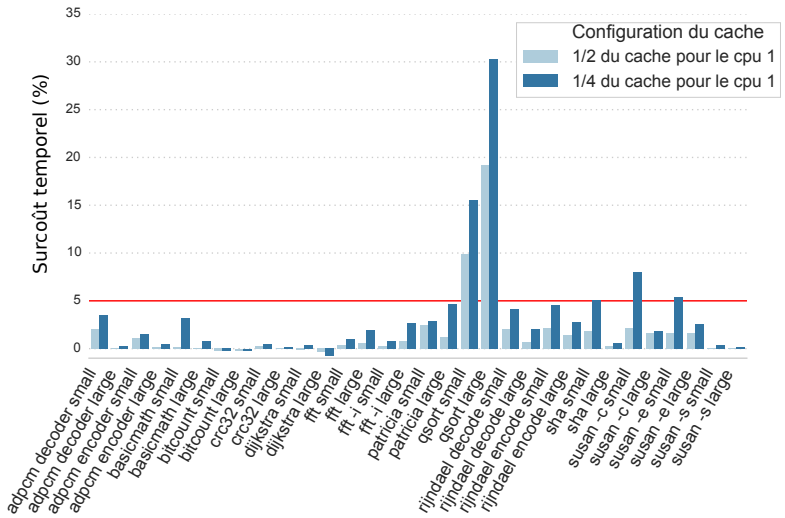


FIGURE 4.7 – Impact du partitionnement du cache L2 sur les performances des appréciations issues de la suite de test MiBench.

Lorsque la partie du cache L2, réservée aux applications MiBench, est réduite à 1/4, nous ob-

servons une dégradation des performances de moins de 5% sur toutes les applications à l'exception de `qsort`, `susan small -c`, et de `susan small -e`. Ces résultats peuvent être expliqués par le fait que `qsort` est une application largement consommatrice de mémoire, la diminution de la taille du cache utilisable l'impact donc fortement. Nous pouvons également noter, que dans le tableau 4.3, les applications `susan small -c` et `susan small -e` ont la plus courte durée de toutes les applications de MiBench et, par conséquence, sont particulièrement sensibles à toute interférence. Les résultats globaux montrent qu'en règle générale les applications de la suite de test MiBench ne sont pas particulièrement consommatrices de mémoire et que leur empreinte mémoire tient majoritairement dans un quart du cache L2.

4.4.2 Impact de la contention mémoire sur les temps d'exécutions

Nous avons ensuite étudié la dégradation de performances qui se produit lorsque le bus mémoire est fortement sollicité. Nous avons lancé les applications de la suite de tests MiBench sur le cœur 0 et une instance de notre programme « charge » sur les trois cœurs restants. Tous les processus sont exécutés en utilisant la politique d'ordonnancement FIFO avec la priorité maximale et sont verrouillés sur leur cœur pour éviter les migrations intempestives.

La figure 4.8 présente les résultats de notre expérience où le surcoût est calculé par rapport aux applications MiBench exécutées isolément et sans partitionnement. Nous pouvons observer que le surcoût maximum est atteint par l'application `qsort large` qui souffre d'un ralentissement de 183%. Nous observons également que toutes les applications, qui souffrent d'un ralentissement temporel, voient son impact réduit par le partage du cache. En effet, les charges étant conçues pour chasser des lignes de caches afin de propager les écritures dans la mémoire principale, et tous les processus partageant le cache de niveau 2, les charges évincent donc les lignes de caches utilisées par les applications de MiBench. Nous pouvons noter, en outre, que les surcoûts temporels mesurés avec la configuration de partitionnement du cache L2 1/4 et la configuration de partitionnement 3/4 varient légèrement. Nous expliquons ses différences par l'impact que le partitionnement du cache a sur le comportent des applications exécutées.

4.5 Conclusion

Dans cette section nous avons, tout d'abord, effectué une étude détaillée de la hiérarchie mémoire de notre carte matérielle, en mettant en évidence, pour chaque composant matériel, les problèmes d'isolation spatiale et temporelle.

Nous avons ensuite sélectionné les configurations matérielles les plus adaptées à un système à criticité multiple dans lequel un éventail d'applications temps réel et *best effort* sont exécutées en parallèle.

Enfin, nous avons évalué l'impact de ces configurations d'une part sur les performances des applications et, d'autre part, sur la réduction des interférences inter-cœurs. Nous avons observé que la technique de partitionnement du cache L2, ici mis en œuvre, a un impact modéré sur le temps d'exécution des applications et ce quel que soit la configuration de partitionnement

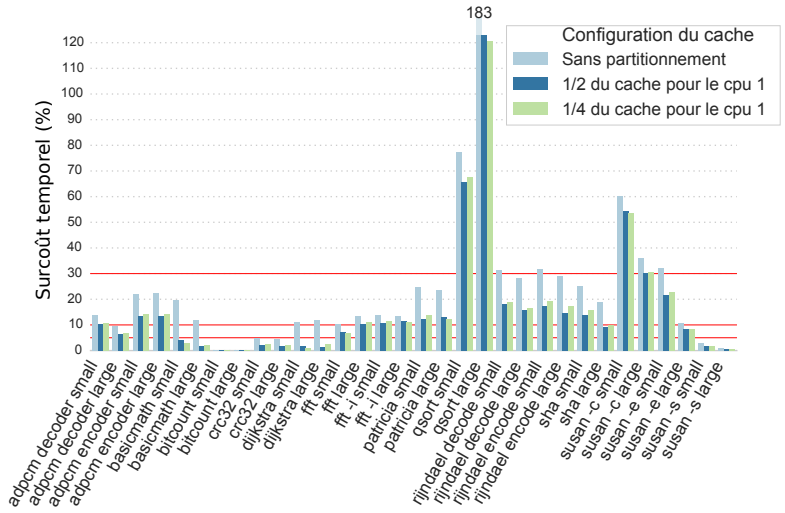


FIGURE 4.8 – Impact des applications « charges » sur les performances des appréciations issues de la suite de test MiBench avec différentes configurations de partitionnement du cache L2.

choisie. Dans la suite de nos travaux, nous utiliserons donc la configuration de partitionnement 1/4-3/4 qui a, pour avantage, de fournir suffisamment d'espace cache pour les applications de *MiBench* tout en maximisant l'espace utilisable par les applications *best effort*.

Nous remarquons, en outre, que si la technique de partitionnement du cache L2 réduit les interférences temporelles inter-programme, elle ne permet pas de les éliminer totalement. En effet, 21 des applications de la suite *MiBench* souffrent d'un surcoût temporel supérieur à 5% et ce même lorsque le partitionnement du cache est activé. Un mécanisme additionnel de gestion des surcoûts temporels générés par la contention mémoire doit donc être mis en œuvre sur notre plate-forme.

Chapitre 5

Une nouvelle approche de contrôle de la mémoire

Sommaire

5.1 Approche . . . 100
5.1.1 Étape hors ligne : caractérisation de la plate-forme . . . 101
5.1.2 Étape en ligne : contrôle . . . 102
5.2 Caractérisation de la plate-forme logicielle et matérielle . . . 103
5.2.1 Phases applicatives . . . 104
5.2.2 Génération de charges mémoires constantes paramétrables . . . 106
5.2.3 Collecte des données . . . 109
5.2.4 Traitement des données et génération des tables . . . 111
5.3 Contrôle à l'exécution . . . 114
5.3.1 Algorithme . . . 115
5.3.2 Mesure de la consommation mémoire . . . 116
5.3.3 Surveillance par échantillonnage . . . 116
5.3.4 Tables embarquées . . . 117
5.3.5 Suspension des applications . . . 118
5.4 Evaluation . . . 118
5.4.1 Surcoût à l'exécution . . . 119
5.4.2 Efficacité pour des charges constantes . . . 120
5.5 Conclusion . . . 127

Nous avons, dans le chapitre précédent, mis en évidence les impacts des interférences temporelles sur des applications temps réel exécutées sur notre carte embarquée en parallèle d'applications *best-effort*.

Dans ce chapitre, nous allons proposer et évaluer une nouvelle approche de contrôle logicielle, adaptée à notre plate-forme matérielle et à nos cas d'utilisation. Nous souhaitons garantir le respect des contraintes temps réel de nos programmes critiques tout en maximisant le parallélisme entre lesdits programmes et les applications *best effort*. Notre approche repose sur la

mise en œuvre d'un mécanisme de contrôle en ligne qui, périodiquement, utilise un oracle pour détecter les surcoûts temporels qui impactent la tâche temps réel ordonnancée et, lorsque les ralentissements deviennent trop importants, suspend les tâches *best effort* afin de permettre à l'application temps réel de terminer son activation sans aucune interférence. La mise en œuvre de notre solution est conditionnée par la présence, dans la plate-forme matérielle, d'un compteur du trafic mémoire, exigence qui est généralement satisfaite sur la plupart des plates-formes multi-cœurs.

Dans la première partie de ce chapitre, nous présenterons plus en détails l'approche que nous avons mise en œuvre pour ensuite, dans une deuxième partie, décrire plus précisément les étapes d'élaboration de l'oracle. Dans un troisième temps, nous détaillerons le mécanisme de contrôle à l'exécution que nous avons développé pour terminer sur une partie dans laquelle nous évaluerons notre solution.

5.1 Approche

Nos travaux ont été développés dans l'optique d'être appliqués sur une machine multi-cœurs où une application temps réel est ordonnancée sur un cœur en parallèle d'applications *best effort* qui sont exécutées sur les cœurs restants. Le nombre maximal de cœurs *best effort* actifs sur la plate-forme matérielle doit être fixé et communiqué au système de contrôle à l'exécution. Ce choix peut être effectué aussi bien en avance, par le concepteur du système qui connaît le nombre de cœurs utilisés par les applications non critiques, que dynamiquement, en utilisant les compteurs matériels de la carte (Cache L1 MISS) pour inférer le nombre de cœurs *best effort* actifs. Nous avons, pour des raisons de simplicité, utilisé la première approche dans nos travaux.

A l'inverse des approches globales qui n'autorisent pas les accès concurrents à la hiérarchie mémoire, l'objectif de notre solution est d'obtenir un gain de parallélisme aussi grand que possible, entre les applications *best effort* et les applications temps réel, aussi longtemps que le surcoût temporel induit par la première classe d'applications sur le deuxième type d'applications reste en dessous d'un seuil spécifié par le concepteur du système.

Pour ce faire, nous supposons que les applications temps réel exécutées sur la plate-forme sont connues par avance, sont périodiques, et peuvent être profilées pour déterminer leur pire temps d'exécution et leur pire consommation de ressources à chaque activation. De plus, nous partons du principe que les applications *best effort* peuvent être démarrées ou stoppées à n'importe quel moment et que leur consommation mémoire nous est inconnue.

Nous allons, dans les parties suivantes, présenter les deux étapes que nous avons mises en œuvre pour atteindre nos objectifs, à savoir, un traitement hors ligne afin de caractériser les impacts de la contention mémoire sur nos applications temps réel et un traitement en ligne pour réguler les accès mémoire effectués par les applications *best effort*.

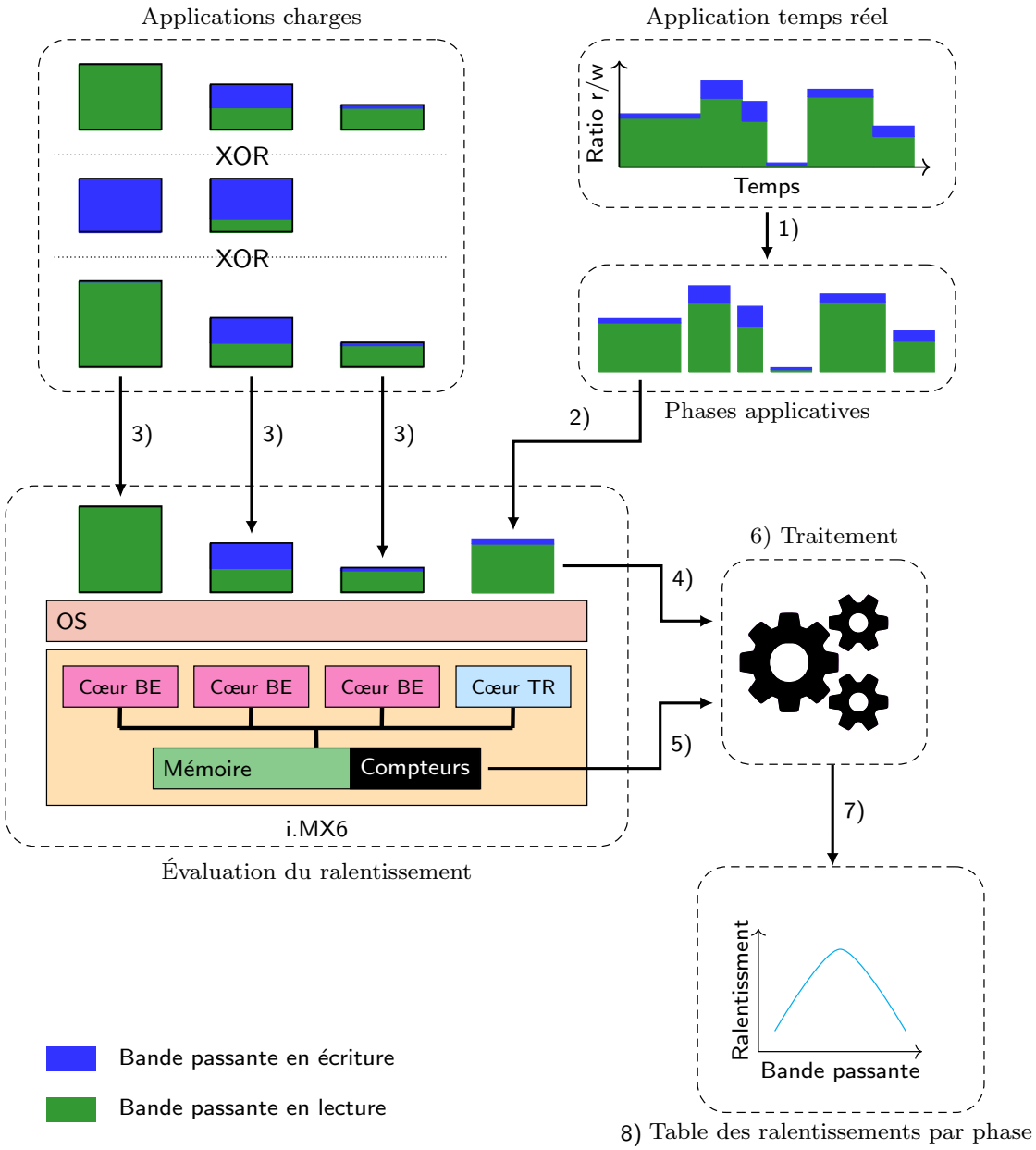


FIGURE 5.1 – Étape hors ligne

5.1.1 Étape hors ligne : caractérisation de la plate-forme

Cette étape consiste en une évaluation de la plate-forme matérielle. Son but est de construire un oracle capable, à partir de la bande passante mesurée auprès du contrôleur mémoire, de formuler une estimation conservatrice des ralentissements qui peuvent impacter une tâche temps réel.

Une des observations clés qui détermine l'élaboration de notre approche est que la capacité de traitement du contrôleur mémoire de notre carte varie en fonction de la proportion de

requêtes en lecture et en écriture qui sont reçues. Ces variations se répercutent sur les ralentissements subis par une tâche temps réel qui dépendent, à la fois, de l'intensité du trafic généré par les différents cœurs et du ratio entre les lectures et les écritures présent au sein du trafic mémoire.

Nous proposons, pour prendre en compte cette observation, une solution, schématisée dans la figure 5.1, afin de caractériser, depuis une bande passante globalement mesurée, les retards pouvant impacter les tâches temps réel.

Dans une première étape **(1)**, réalisée hors ligne par le concepteur du système, nous effectuons une analyse manuelle du code source des applications temps réel pour identifier des **phases** applicatives durant lesquelles une application exécute un pattern de code répétitif qui génère un trafic mémoire homogène (Taux de lectures et d'écritures).

Le concepteur du système exécute ensuite chaque phase de l'application temps réel **(2)** contre une variété de charges **(3)** afin de mesurer, auprès des compteurs matériels, le nombre d'accès mémoire **(5)** et le temps d'exécution **(4)** de chacune des phases obtenant ainsi, pour chaque phase, la bande passante moyenne mesurée et le ralentissement temporel induit. Nous avons, pour mener à bien cette étape d'analyse, étendu le *microbenchmark* charge, décrit précédemment en section 4.3.4, afin de générer divers trafics mémoire qui varient aussi bien en termes de bande passante qu'en termes de taux de lecture et d'écriture.

Les données ainsi collectées sont traitées hors ligne **(6)** afin de générer **(7)** une **table des ralentissements (8)** qui, pour une phase donnée et pour un nombre de cœurs *best efforts* actifs déterminé, associe à une bande passante mesurée un surcoût temporel.

5.1.2 Étape en ligne : contrôle

Après le traitement hors ligne, effectué une fois par application, nous mettons en place un mécanisme de contrôle, présenté dans la figure 5.2, activé à chaque exécution des applications. Avant le démarrage des applications temps réel, le composant intitulé **régulateur (1)**, intégré dans le système d'exploitation ou dans l'hyperviseur, est activé et les applications temps réel sont alors exécutées en parallèle des applications *best effort*.

Le régulateur va périodiquement échantillonner les compteurs matériels **(2)** du contrôleur mémoire pour récupérer la bande passante générée par l'ensemble de la couche logicielle. Les valeurs ainsi obtenues sont utilisées en entrée de la table des surcoûts **(3)** afin d'obtenir une estimation conservatrice des ralentissements **(4)**, générés par les interférences sur le système mémoire, qui ont impactés la tâche temps réel.

Si le surcoût total estimé, depuis le début de l'exécution de l'application, est supérieur au seuil fixé par le concepteur du système, moins le surcoût temporel maximal qui peut être accumulé en un seul échantillonnage **(5)**, le système de contrôle à l'exécution **(6)** suspend toutes les applications *best-effort*. L'application temps réel peut alors terminer son exécution courante sans souffrir de ralentissements additionnels. Les applications suspendues sont redémarrées **(8)** lorsque l'application temps réel termine son activation.

Autrement **(7)**, les applications *best effort* continuent leurs exécutions jusqu'à la prochaine période d'activation du régulateur.

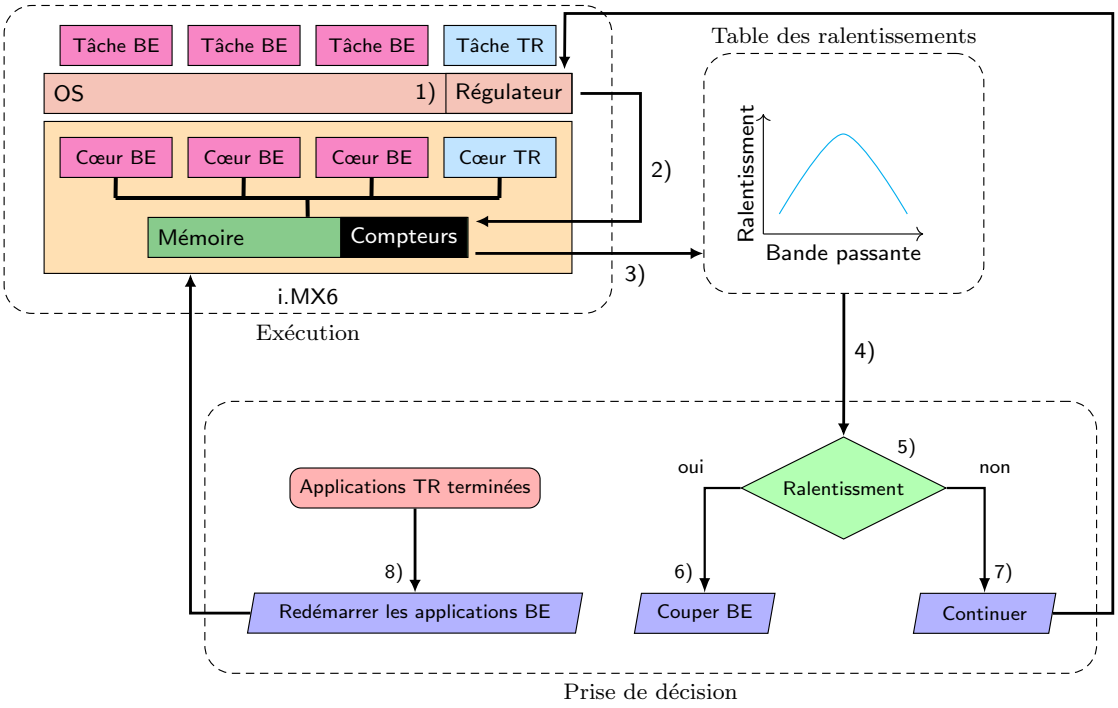


FIGURE 5.2 – Étape en ligne

Dans la suite de cette partie, nous allons décrire les diverses analyses et mécanismes utilisés pour mettre en œuvre notre approche. Toutes les expérimentations effectuées ont désormais été réalisées avec le cache L2 partitionné, de telle sorte à ce que nous puissions porter notre attention sur la gestion de la contention au niveau du sous-système mémoire. Nous avons utilisé la configuration de partitionnement 1/4 - 3/4 qui fournit un espace de mémoire cache suffisant pour les applications de MiBench tout en maximisant l'espace disponible pour les applications *best-effort*.

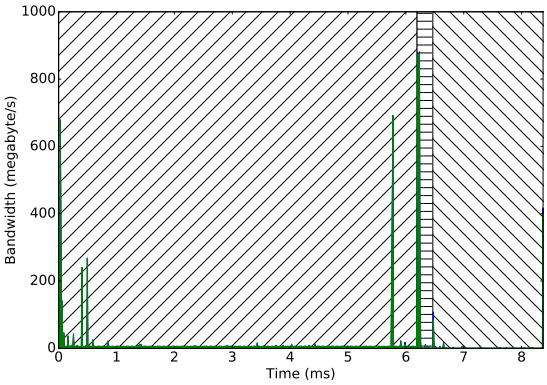
5.2 Caractérisation de la plate-forme logicielle et matérielle

Après avoir présenté une vue globale de notre solution, nous allons maintenant étudier, plus en profondeur, le processus hors ligne permettant de caractériser les impacts de la contention pour une plate-forme logicielle et matérielle donnée.

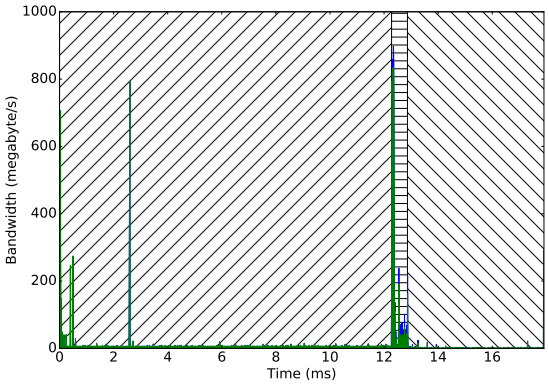
Dans une première section, nous allons étudier le découpage des applications temps réel en de multiples phases pour, dans un deuxième temps, décrire les modifications que nous avons appliquées sur les *microbenchmark* charge développés dans le chapitre précédent. Nous décrirons, ensuite, la configuration des multiples expériences que nous avons effectuées sur notre carte pour récupérer les données brutes qui caractérisent le comportement de nos applications temps réel. Enfin, nous détaillerons le processus de traitement des données récupérées antérieurement pour générer notre table de prédiction des surcoûts temporels.

5.2.1 Phases applicatives

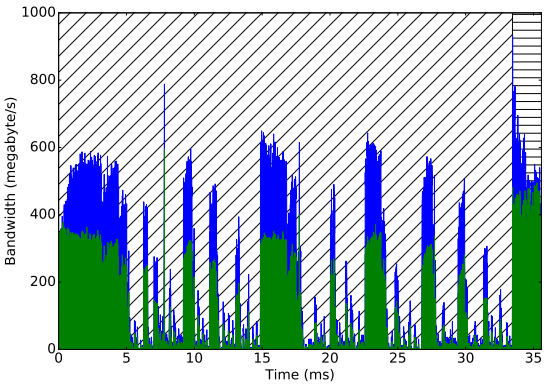
Nous avons identifié, par une analyse manuelle du code des applications temps réel, les portions du code source dans lesquelles les applications exécutent un pattern de code répétitif (Boucles) qui émet un trafic mémoire récurrent. Nous avons ensuite modifié les applications multi-phasées pour ajouter une mesure du temps d'exécution et de la bande passante en utilisant les compteurs du matériel.



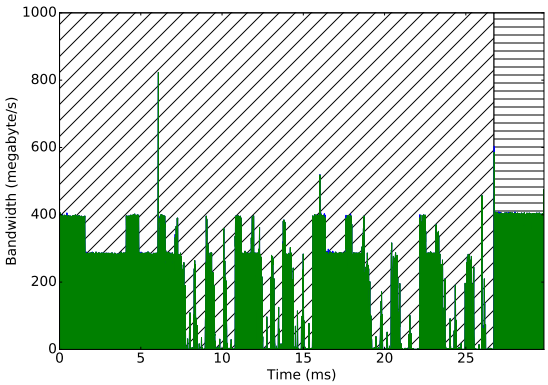
(a) Fft small (3 phases)



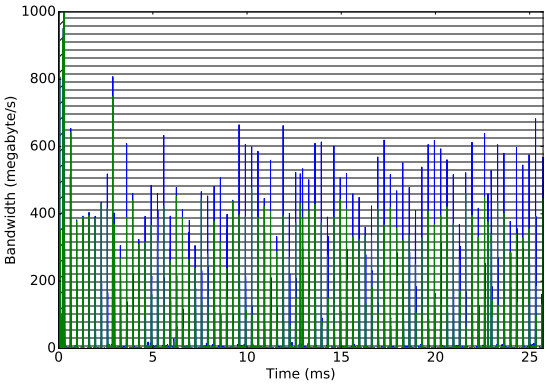
(b) Fft small -i (3 phases)



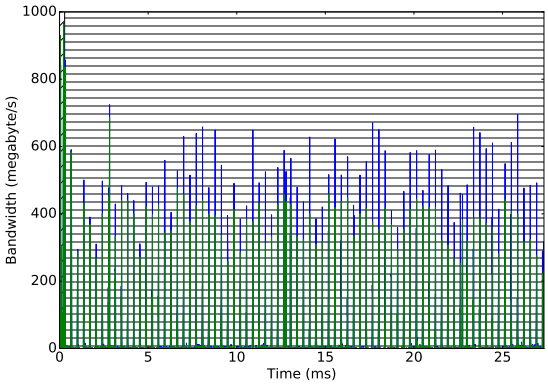
(c) Qsort small (2 phases)



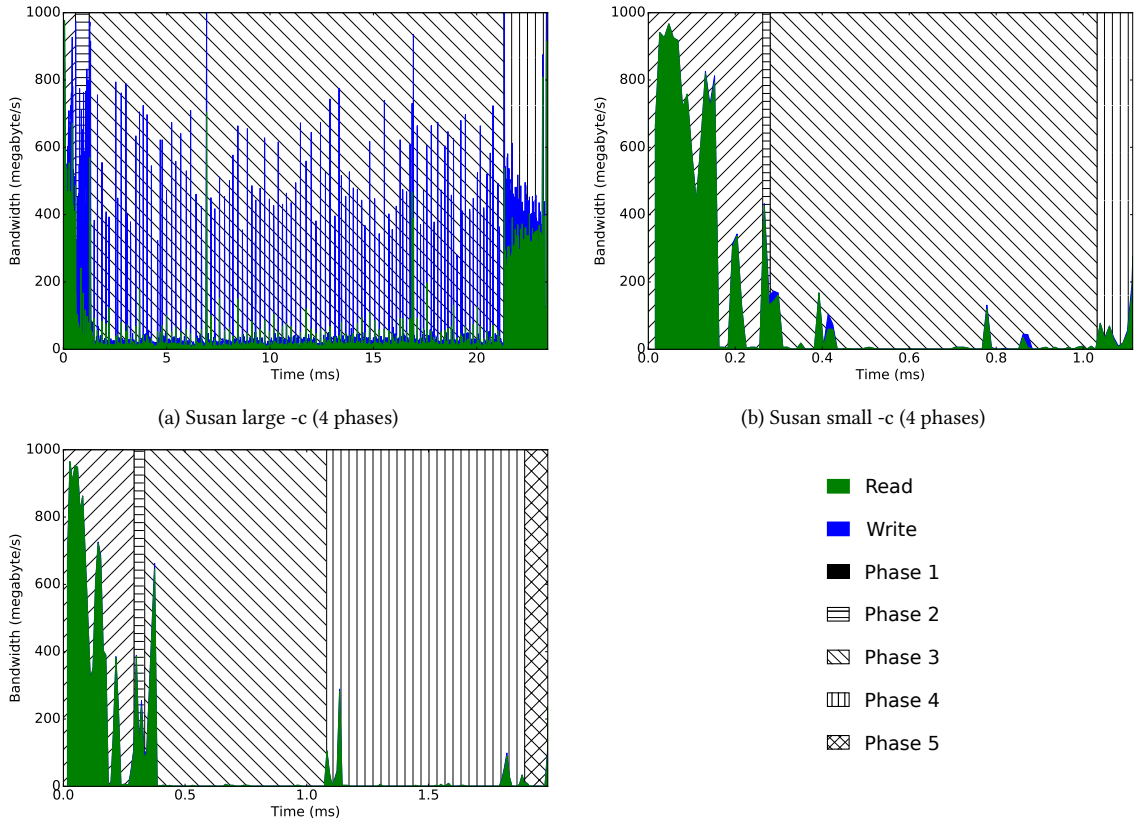
(d) Qsort large (2 phases)



(e) Rijndael small -d (2 phases)



(f) Rijndael small -e (2 phases)



(a) Susan large -c (4 phases)

(b) Susan small -c (4 phases)

(c) Susan small -e (5 phases)

FIGURE 5.4 – Profils mémoire des applications ayant des phases applicatives sélectionnées au sein de la suite MiBench

La figure 5.4 affiche les profils mémoire des applications sélectionnées dont le code source comporte plusieurs phases. Chaque sous-figure, réalisée en utilisant le profileur décrit dans la section 5.3.3, présente le profil mémoire d'une exécution de l'application. Le trafic effectué vers le contrôleur mémoire en écriture est affiché en bleu alors que celui effectué en lecture est en vert, les deux trafics étant affichés en cumulatif.

En confrontant l'analyse des phases du code source aux profils mémoire nous avons observé une correspondance entre les motifs d'accès au code et les phases mémoires. Nous observons cependant que des variations de bande passante subsistent au sein des phases applicatives tel que dans l'application `qsort`. Nous expliquons ces variations comme étant un effet de bord des caches, une même instruction exécutée de manière répétitive ne provoque pas le même comportement mémoire selon l'état du cache.

Nous avons cependant jugé que le modèle d'estimation des ralentissements temporels reste valide aussi longtemps que le ratio de lecture-écriture au sein de chaque phase reste constant. Nous constatons également que les phases applicatives de certaines applications (première phase de `rijndael` et la deuxième phase de `susan small`) ont une faible durée d'exécution.

5.2.2 Génération de charges mémoires constantes paramétrables

Nous allons maintenant étudier le programme charge, précédemment développé dans la partie 4.3.4 pour mettre en évidence le problème d'interférences mémoire sur notre architecture matérielle, que nous avons modifié pour caractériser le comportement de nos programmes temps réel face à des interférences mémoire variables.

Dans une première partie, nous détaillerons les modifications du code source que nous avons réalisées pour, ensuite, effectuer une étude du comportement mémoire du programme ainsi modifié.

5.2.2.1 Code source

Si le *microbenchmark*, précédemment décrit dans la partie 4.3.4, permet d'étudier les interférences temporelles générées par une charge constante et maximale en écriture sur des applications temps réel, il ne permet pas de caractériser les ralentissements générés par un trafic mémoire en lecture ou par des charges moins intensives. Nous l'avons donc étendu, d'une part, afin de générer un trafic mémoire mixte contenant à la fois des lectures et des écritures et, d'autre part, pour pouvoir faire varier la bande passante générée. La figure 5.5 contient le code source de notre programme qui, ainsi modifié, prend en entrée, en sus d'un pointeur sur le tableau parcouru, le nombre total d'itérations à effectuer sur le tableau, le nombre de lectures, le nombre d'écritures et le nombre de tours de boucle d'attente à réaliser entre les accès à la mémoire.

La boucle interne, identifiée par le label `rw_stress_loop`, va désormais parcourir le tableau en appliquant, de manière répétitive, le pattern décrit ci-après consistant en une succession d'écritures, à la granularité d'une ligne de cache, avec la boucle `rw_write_loop` suivi d'une succession de lectures, effectuées dans la boucle `rw_read_loop`, pour ensuite, dans la boucle `rw_delay_loop`, effectuer une attente active sans aucunement générer de consommation mémoire. La boucle externe `rw_outer_loop` est, quant à elle, utilisée pour spécifier le temps d'exécution de la charge en contrôlant le nombre d'itérations totales faites sur le tableau. En réglant le nombre de lectures et d'écritures réalisées respectivement dans les boucles `rw_write_loop` et `rw_read_loop` nous pouvons faire varier le type de trafic émis par notre charge tandis que la boucle `rw_delay_loop` nous permet de faire varier l'intensité du trafic généré.

```
1  .arm
2  .text
3  .align
4
5  .global stress_read_write
6  stress_read_write:
7      push {r4-r12,lr}        @ Sauvegarde des registres sur la pile
8
9      ldmia r0, {r0-r5}       @ r0 -> Adresse du tableau
10                             @ r1 -> Taille du tableau en octets
11                             @ r2 -> Nombre d'iteration sur le tableau
12                             @ r3 -> Nombre de lignes de caches a ecrire
13                             @ r4 -> Nombre de lignes de caches a lire
14                             @ r5 -> Delais entre les lecture et les ecritures
15     mov r11, #0             @ r11 et r12 contiennent les valeurs ecrites en memoire
16     mov r12, #0
17     mov lr, r3              @ lr -> Nombre d'ecritures
18     add lr, r4              @ lr -> Nombre d'ecritures + Nombre de lectures
19     lsl lr, #5              @ lr -> Taille totale en octets a parcourir
20
21 rw_outer_loop:              @ Label de la boucle pour iterer plusieurs fois sur le
22                             @ tableau
23     mov r6, r0              @ r6 <- Iterateur du tableau
24     mov r7, r1              @ r7 <- Taille restante a iterer
25
26 rw_stress_loop:             @ Label de la boucle qui itere sur le tableau
27     mov r8, r3              @ r8 -> Nombre d'ecritures restantes a effectuer
28     mov r9, r4              @ r9 -> Nombre de lectures restantes a effectuer
29     mov r10,r5              @ r10 -> Delais
30
31     subs r7, lr             @ Si il ne reste pas assez de place dans le tableau pour
32     ble rw_end              @ effectuer les lecture et ecritures voulues
33                             @   alors branchement vers rw_end
34
35 rw_write_loop:
36     subs r8, #1             @ Decremente le nombre d'ecritures restantes
37     stmgeia r6!, {r11,r12}  @ Ecrit 8 octets et incremente l'iterateur en consequence
38     stmgeia r6!, {r11,r12}
39     stmgeia r6!, {r11,r12}
40     stmgeia r6!, {r11,r12}  @ 32 octets (1 ligne de cache) ont alors ete ecrits
41     bgt rw_write_loop       @ Si toutes les ecritures n'ont pas ete faites alors
42                             @ branchement vers rw_write_loop
43
44     mov r12, #0             @ Remise a zero de r11 et de r12
45     mov r11, #0             @ pour eviter les overflows
46
47 rw_read_loop:
48     subs r9, #1             @ Decremente le nombre de lectures restantes
49     ldmgeia r6!, {r11,r12}  @ Lit 8 octets et incremente l'iterateur en consequence
50     ldmgeia r6!, {r11,r12}
51     ldmgeia r6!, {r11,r12}
52     ldmgeia r6!, {r11,r12}  @ 32 octets (1 ligne de cache) ont alors ete lus
53     bgt rw_read_loop        @ Si toutes les lectures n'ont pas ete faites alors
54                             @ branchement vers rw_read_loop
55
56 rw_delay_loop:
57     subs r10, #1            @ Incremente la boucle de delais et les registres
58     add r11, #1             @ r11 et r12
59     add r12, #1
60     bgt rw_delay_loop       @ Si tous les delais n'ont pas ete effectues
61                             @    alors branchement vers rw_delay_loop
62     b rw_stress_loop        @ Branchement vers la boucle qui itere sur le tableau
63
64 rw_end:
65     subs r2, #1             @ Decremente le nombre de fois ou l'on doit iterer sur
66                             @ le tableau
67     bgt rw_outer_loop       @ Reboucle si il reste des iterations a effectuer
68     pop {r4-r12,pc}         @ Restauration des registres de piles
```

FIGURE 5.5 – Boucle principale du programme charge modifié pour solliciter la hiérarchie mémoire avec des bandes passantes et des taux de lecture et d'écriture variables

5.2.2.2 Étude expérimentale des charges

Dans cette partie, nous allons évaluer expérimentalement, aussi bien en termes de bande passante mémoire générée que de taux de lecture et d'écriture réalisés, les impacts de notre *microbenchmark* modifié sur la hiérarchie mémoire de notre architecture matérielle.

Nous avons donc exécuté de multiples fois notre programme charge en isolation sur un cœur dédié de notre carte, les cœurs restants étant désactivés, et nous avons mesuré le trafic mémoire auprès des compteurs matériels du contrôleur mémoire. Dans cette expérience, nous avons fait varier aussi bien le taux de lecture et d'écriture que le nombre de tours de boucle d'attente active.

Les bandes passantes totales résultantes de ces expériences sont présentées dans la figure 5.6, tandis que la figure 5.7 contient le trafic mesuré en lecture et la figure 5.8 affiche celui observé en écriture. Nous avons utilisé une échelle logarithmique [1] en abscisse pour afficher le nombre de tours de boucle d'attente active entre les boucles effectuant des lectures et celles effectuant des écritures, tandis que l'ordonnée représente la bande passante mesurée en gigaoctets par secondes. Les multiples courbes modélisent les variations de lecture et d'écriture entre les charges sous la forme `xr-yw` ou `x` indique le nombre d'itérations effectuées dans la boucle `rw_read_loop` et `y` le nombre de tours de la boucle `rw_write_loop`.

Le trafic mémoire maximal observé au sein de nos expériences est atteint par la configuration `0r-10w` avec une bande passante en écriture de 2020 MB/s (Figure 5.8) en l'absence de trafic généré en lecture. La bande passante maximale en lecture est, quant à elle, logiquement générée par la configuration `10r-0w` qui atteint un maximal de 425 MB/s (Figure 5.7) aucun trafic en écriture n'étant généré par la charge. Nous expliquons la différence entre le trafic maximal atteint par une charge en écriture et celui atteint en lecture par le fait, qu'à contrario des requêtes en lecture, celles en écriture sont non-bloquantes, ce qui permet à la charge d'être plus intensive.

L'utilisation d'un mixte de requêtes en lecture et en écriture induit une compétition pour l'accès au tampon de réordonnancement du contrôleur mémoire entraînant un effondrement du trafic en écriture. En effet, nous observons que la bande passante en écriture de la configuration `1r-9w` qui introduit une seule lecture supplémentaire par rapport à la configuration `0r-10w` est divisée par plus de 2,5 fois. L'ajout de lectures additionnelles (Configurations `1r-9w` à `9r-1w`) provoque une dégradation moins forte mais toujours présente du trafic en écriture qui n'est pas compensée par l'augmentation de la bande passante en lecture diminuant au final le total de la bande passante mesurée.

L'impact des délais, sur les bandes passantes observées, se traduit par une progression selon une loi exponentielle décroissante. Le trafic mémoire affiché dans la figure 5.6 pour la configuration `0r-10w` décroît d'abord fortement de 2 GiB/s à 150 MiB/s lorsque les délais augmentent de 0 à 1000 pour ensuite baisser plus légèrement jusqu'à atteindre 40 MiB/s quand les délais atteignent 4000 itérations, la bande passante mesurée se stabilisant aux alentours de 30 MiB/s pour des valeurs de délais supérieures à 4000. Ce comportement se retrouve également dans les autres configurations de lectures et d'écritures mais aussi lorsque l'on regarde individuel-

1. Pour afficher le délai 0 malgré l'utilisation d'une échelle logarithmique, nous avons décalé le point 0 en 0,11

lement les bandes passantes en lecture et en écriture respectivement exposées dans les figures 5.7 et 5.8.

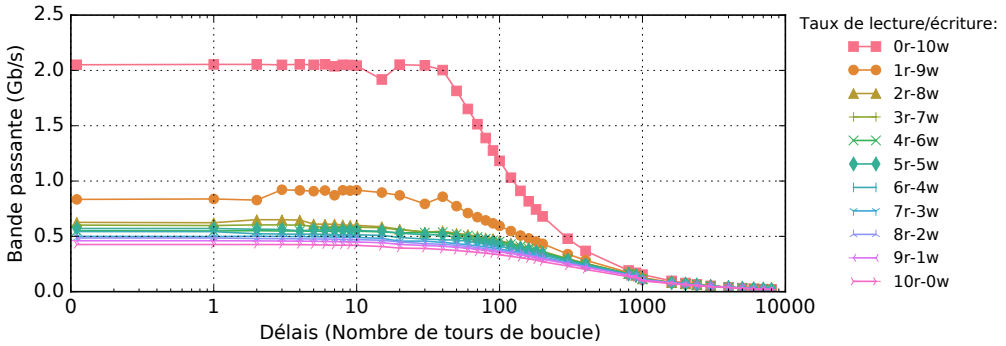


FIGURE 5.6 – Bande passante totale d'une charge exécutée en isolation

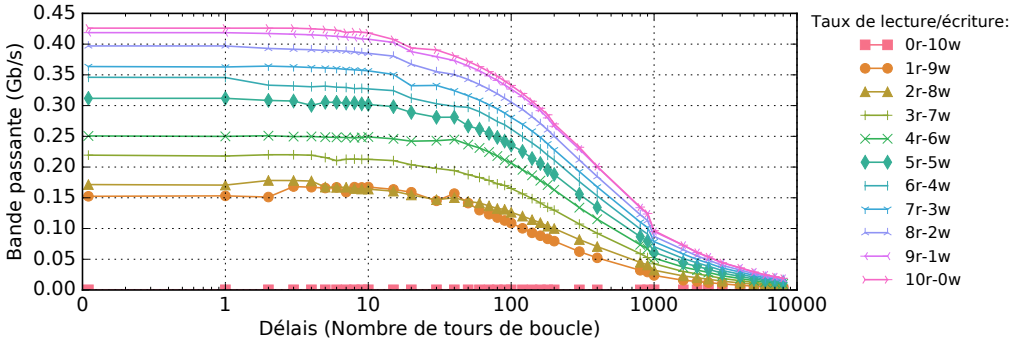


FIGURE 5.7 – Bande passante en lecture d'une charge exécutée en isolation

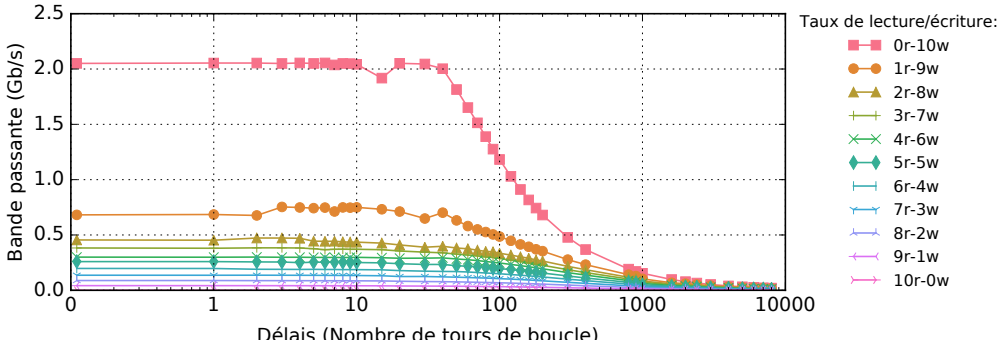


FIGURE 5.8 – Bande passante en écriture d'une charge exécutée en isolation

5.2.3 Collecte des données

Nous allons dans cette étape, décrire les différentes expériences que nous avons effectuées pour obtenir les données brutes, qui caractérisent le comportement d'une application temps réel face à des interférences temporelles et que nous allons ensuite post-traiter pour générer les tables de ralentissements.

La première expérience que nous avons réalisée a pour objectif de récupérer des informations sur le comportement des applications temps réel en l'absence de contention. Nous avons donc exécuté en isolation chaque application temps réel précédemment sélectionnée, pour mesurer ensuite, pour chaque phase applicative p, le pire temps d'exécution observé $ExecT_p$ ainsi que le nombre d'accès mémoires Acc_p mesurés par les compteurs matériels au sein de la phase.

La deuxième expérience devait permettre de collecter des données sur le ralentissement des programmes temps réel issu des interférences générées par des charges ordonnancées en parallèle. Nous avons donc exécuté plusieurs fois chaque application temps réel en parallèle de multiples configurations de charges constantes dénommées l.

Notre oracle devant, à terme, être capable d'estimer, depuis une bande passante mesurée, les ralentissements qui peuvent impacter les phases applicatives des applications temps réel. Nous avons donc modulé la bande passante générée par les *microbenchmark* en ajoutant des tours de boucle d'attente d'active allant de 0 itération, générant ainsi une bande passante maximale, à 8000 itérations le trafic mémoire généré pour de telles valeurs étant quasiment nul.

Pour obtenir de meilleures performances notre solution de contrôle prend en compte le nombre de cœurs *best effort* actifs. En effet, les interférences qui peuvent ralentir une application temps réel sont moindres lorsque un seul cœur *best effort* est actif que lorsque deux cœurs le sont, le trafic mémoire généré par une seule application *best effort* étant inférieur. Nous avons donc pris en compte ce comportement en ordonnançant de une à trois charges en parallèle sur les trois cœurs non temps réel de la carte.

Nous avons, pour chaque configuration, fait varier aussi bien le nombre de charges exécutées que la bande passante et le ratio de lecture et d'écriture généré par chacune d'entre elles, le but étant de caractériser l'impact de telles variations sur les temps d'exécution des applications temps réel et sur la bande passante mesurée.

Nous avons également étudié l'impact des interférences temporelles générées par des charges ayant des bandes passantes symétriques et asymétriques. Dans la configuration asymétrique à deux cœurs, le nombre de tours de boucle d'attente de la deuxième charge est égal à deux fois celui de la première charge tandis que dans la configuration asymétrique à trois cœurs le nombre de tours de boucle du troisième cœur est égal à trois fois celui du premier cœur.

De plus, comme nous l'avons vu dans la section 5.2.2.2, les taux de lecture et d'écriture générés par les programmes ont un impact sur la bande passante qui peut être traitée par le contrôleur et donc, in fine, sur les ralentissements pouvant impacter une tâche. Nous avons donc fait varier le ratio de lecture et d'écriture des accès mémoire effectués par les charges respectivement dans les boucles `rw_read_loop` et `rw_write_loop` de telle sorte à ce que la somme du nombre de lignes de cache lues et écrites dans chacun des tableaux soit égale à dix. Une telle configuration permet d'obtenir une bonne diversité quant aux taux de lecture et d'écriture présents dans les bandes passantes générées.

Au final, nous avons collecté les mêmes métriques que celles mesurées dans la première expérience à savoir que, pour chaque configuration de charge l et chaque phase p de la tâche temps réel, nous mesurons le trafic mémoire $Acc_{l,p}$ globalement généré par l'ensemble des applications exécutées et le temps maximal d'exécution $ExecT_{l,p}$ de la phase p.

5.2.4 Traitement des données et génération des tables

Dans cette section, nous allons détailler le processus de post-traitement que nous avons appliqué sur les données collectées dans l'étape précédente pour générer les tables qui, depuis une bande passante mesurée, retournent une estimation du ralentissement imputé à une tâche temps réel.

Dans la première étape de ce traitement nous avons calculé, pour chaque phase p de la tâche temps réel exécutée en parallèle des l charges, la bande passante observée, $ObsB_{l,p}$, comme étant égale aux accès mémoire mesurés dans la phase p exécutée en parallèle de charges l, noté $Acc_{l,p}$, divisée par le temps d'exécution de la phase p exécutée en parallèle de charges l dénommé $ExecT_{l,p}$ soit $ObsB_{l,p} = Acc_{l,p}/ExecT_{l,p}$.

Nous avons également calculé, pour chacune des p phases de l'application temps réel, le ralentissement $Ovd_{l,p}$ induit par les interférences temporelles générées par les charges l comme étant égal au ratio entre le temps d'exécution de la phase p exécutée en parallèle des charges l ($ExecT_{l,p}$) et le temps d'exécution de la phase p exécutée seule $ExecT_p$ soit $Ovd_{l,p} = (ExecT_{l,p}/ExecT_p) - 1$.

Le résultat de cette première étape est un ensemble de clés valeurs associant à chaque bande passante observée $ObsB_{l,p}$ un surcoût temporel estimé $Ovd_{l,p}$ pour chaque phase p des applications temps réel. Nous notons, en plus, pour chaque association de l'ensemble, le nombre de processus de type charge utilisé, le ratio de lecture-écriture mis en œuvre par les charges et le nombre de tours de boucle d'attente utilisé dans les charges pour générer la charge l.

Les multiples bandes passantes observées ($ObsB_{l,p}$), ont été calculées pendant l'étape de collecte des données en utilisant les accès mémoire effectués (Acc_p) dans chaque phase applicative, la valeur des compteurs matériels du contrôleur mémoire étant relevée au début et à la fin de chaque phase. Toute variation, même minime, des accès au sein d'une phase applicative, est donc lissée et moyennée au sein de la bande passante calculée sur la totalité de la phase. Par conséquent, les associations entre les bandes passantes observées et les ralentissements qui sont contenues au sein de l'ensemble résultant de la troisième étape du processus de collecte des données ne couvrent pas l'ensemble des bandes passantes qui peuvent être effectivement observées si l'on effectue des mesures ponctuelles au sein d'une phase.

Pour pouvoir estimer les ralentissements induits par des bandes passantes arbitraires pouvant être mesurées au sein des phases, nous avons décidé d'extrapoler les données manquantes, depuis les valeurs de ralentissement mesurées, en utilisant une approche par **interpolation polynomiale**, par la méthode des moindres carrés, tel que fournie par la fonction `polyfit` de la bibliothèque `numpy`. [2]. La mise en œuvre d'une telle démarche implique de relever deux défis.

Tout d'abord, l'utilisation d'une approche par interpolation polynomiale nécessite de choisir le degré approprié pour le polynôme. Ensuite, la technique d'interpolation polynomiale génère, par construction, une courbe qui passe aussi près que possible de l'ensemble des points de ralentissements existants. Or, dans notre situation, nous voulons, générer une estimation conservatrice des surcoûts temporels dont la courbe dérivée passe au-dessus de l'ensemble des points de ralentissements existants.

2. http ://www.numpy.org/

5.2.4.1 Choix du degré du polynôme

Pour choisir le degré du polynôme nous avons mis en œuvre une approche par force brute dans laquelle nous générons de multiples interpolations, en utilisant divers degrés pour le polynôme pour, ensuite, par simulation, évaluer lesquelles nous donnent les meilleurs résultats.

Nous avons donc conçu un simulateur pour évaluer la pertinence de notre solution de contrôle à l'exécution des applications. Ce programme prend en paramètre deux éléments d'entrée, à savoir : une table des surcoûts temporels estimés à partir de la bande passante mesurée par échantillonnage et une trace de la consommation mémoire générée par une application temps réel co-exécutée avec des charges.

En utilisant ces deux éléments, notre simulateur est capable d'estimer le ralentissement subi par la tâche temps réel et par conséquent la date à laquelle les applications *best effort* doivent être suspendues pour que l'application critique respecte ses contraintes temporelles. Nous avons donc effectué de multiples simulations sur de nombreuses traces d'exécutions pour de multiples charges en utilisant des tables de surcoûts générées par interpolation polynomiale et en faisant varier le degré du polynôme entre 1 et 5, l'utilisation de degrés plus élevés générant des comportements erratiques caractéristiques des polynômes de hauts degrés. Nous avons utilisé le résultat de ces simulations pour choisir le degré qui donne le niveau d'erreur le plus faible mesuré en utilisant la somme du carré des résidus, entre les ralentissements estimés et ceux mesurés pour le nombre de ratios lecture-écriture le plus large.

5.2.4.2 Estimation conservatrice

La technique d'interpolation polynomiale génère une courbe qui passe aussi près que possible de l'ensemble des points du jeu de données utilisé par la fonction d'interpolation.

Pour construire un polynôme qui trace les ralentissements les plus importants, nous avons appliqué notre fonction d'interpolation sur chaque jeu de données ayant un taux de lecture et d'écriture communs et pour de multiples délais. Ensuite, pour chaque bande passante, nous avons choisi la valeur maximale au sein des multiples valeurs obtenues.

Si la méthodologie, décrite ci-dessus, permet d'effectuer une approximation des ralentissements subis par une tâche temps réel exécutée en parallèle de charges qui génèrent une bande passante constante, elle ne permet pas de prendre en compte les ralentissements générés par des charges dont la bande passante varie. Afin de généraliser notre approche, nous avons pris en considération le cas où l'intensité de la bande passante générée par les charges change au milieu d'un échantillon de contrôle. Nous prenons comme cas d'étude, l'exemple suivant où la bande passante générée par des charges, pendant le premier quart d'une période d'échantillonnage, atteint les 400 MB/s pour ensuite, dans le temps restant, plafonner à 300 MB/s, le processus de contrôle mesurant au final une bande passante totale de 325 MB/s.

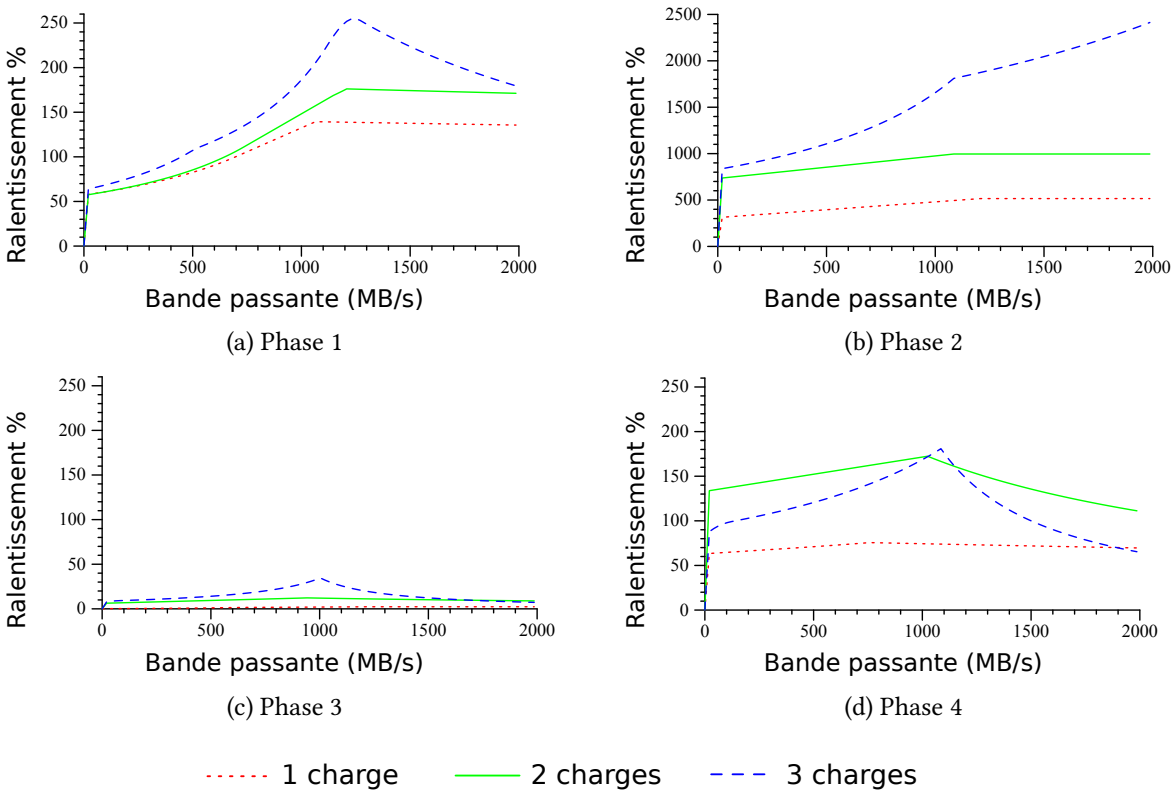
Nous avons donc mis en œuvre une stratégie dite de **packing** qui permet d'estimer le pire ralentissement pouvant être induit par les charges variables à l'intérieur de chaque intervalle d'échantillonnage. Nous sommes donc parti du principe que la bande passante observée $ObsB$ au sein d'un échantillon peut être décomposée en deux bandes passantes nommées $ObsB_1$ et

$ObsB_2$, chacune d'entre elles étant émise durant une fraction de l'intervalle d'échantillonnage nommée t_1 et t_2 où $0 \leq t_1, t_2 \leq 1$, de telle sorte que $t_1+t_2 = 1$ et $t_1 \cdot ObsB_1 + t_2 \cdot ObsB_2 = ObsB$. Nous faisons varier les bandes passantes $ObsB_1$ et $ObsB_2$ entre 0 et 3000 MB/s par pas de 20,48 MB/s, ce qui correspond à la granularité mise en œuvre dans les tables d'estimation des ralentissements utilisées par notre mécanisme de régulation à l'exécution décrit dans la partie 5.3.4, de telle sorte que $ObsB_1 < ObsB_2$ et que les valeurs de t_1 et de t_2 restent dans la plage définie.

Nous avons estimé le surcoût associé à chaque paire de bande passante possiblement observée au sein d'un échantillon de la manière suivante. Nous utilisons les résultats de l'analyse polynomiale, qui, dans le cadre d'une application temps réel exécutée en parallèle de charges constantes, nous permettent d'associer pour chacune des bandes passante observées, $ObsB_1$ et $ObsB_2$, un ralentissement estimé. Afin de déterminer l'impact de la combinaison des bandes passantes au sein d'un unique échantillon, nous observons que le ralentissement, calculé en termes de temps d'exécutions, de la tâche temps réel au sein d'un échantillon réfléchi le ratio entre la quantité de bande passante, nommée *Req*, demandée par l'application temps réel au sein d'une période d'échantillonnage et la quantité de bande passante réellement obtenue, nommée *ObtB*, par ladite application. *Req* représente la bande passante observée lorsque l'application temps réel est exécutée seule et la bande passante obtenue, *ObtB*, peut être calculée depuis le ralentissement observé, *OvdB*, pour une bande passante constante observée, *ObsB*, tel que $ObtB = Req/(OvdB + 1)$. Nous pouvons ensuite, à partir de la bande passante obtenue, calculer le ralentissement induit dans le cas où la bande passante globalement observée se décompose en deux sous bandes passantes, $ObsB_1$ and $ObsB_2$, tel que $Ovd = Req/(t_1 \cdot ObtB_1 + t_2 \cdot ObtB_2)$. Nous prenons au final, pour remplir la table des surcoûts, le ralentissement maximum pour toutes les combinaisons de bandes passantes effectuées.

5.2.4.3 Résultats

La figure 5.9 contient les quatre tables de surcoûts qui ont été calculées pour les phases de `susan small -c`. Nous pouvons remarquer que la deuxième phase, celle ayant la durée la plus courte, prédit, pour toutes les bandes passantes présentes, un ralentissement plus de dix fois supérieur à celui des autres phases. Nous observons également que, pour toutes les phases applicatives, le ralentissement prédit augmente graduellement au fur et à mesure que la bande passante mesurée croît jusqu'à atteindre un maximal de 1000 MB/s après lequel le ralentissement estimé s'effondre. Nous expliquons cet effondrement par le fait qu'une telle bande passante mesurée sur le système ne peut être atteinte que lorsque la contention mémoire diminue, l'application temps réel étant alors moins ralentie par les applications *best effort*. Nous pouvons également noter que la bande passante moyenne observée pour chaque phase est légèrement inférieure à celle qui correspond au point maximal de ces courbes, mais que, en pratique, lorsque le mécanisme de contrôle est activé, des bandes passantes plus importantes peuvent être observées.



(a) Phase 1

(b) Phase 2

(c) Phase 3

(d) Phase 4

FIGURE 5.9 – Tables des ralentissements des quatre phases de `susan small -c`

5.3 Contrôle à l'exécution

Nous allons, dans cette section, détailler le composant régulateur que nous avons développé pour effectuer un **contrôle en ligne** des applications *best effort*. Notre régulateur va mesurer le trafic mémoire émis par les applicatifs pour **détecter**, en utilisant les tables des ralentissements générées à l'étape précédente, les surcoûts temporels induits par la contention mémoire sur les tâches temps réel. Lorsque le total des ralentissements dépasse un seuil fixé par le concepteur du système, une étape de **recouvrement** est mise en œuvre, les tâches *best effort* sont alors suspendues afin d'éliminer les interférences temporelles préservant ainsi le respect des échéances des tâches temps réel.

L'activation des compteurs matériels de mesure de la bande passante nécessitant des droits d'accès privilégiés au matériel, nous avons développé notre profileur sous la forme d'un **module noyau** intégré au sein du système d'exploitation Linux que nous utilisons dans nos expériences. Un module noyau se présente sous la forme d'un fichier objet contenant du code machine pouvant être chargé dynamiquement par un système d'exploitation. Le module est alors exécuté en mode superviseur ce qui lui permet d'interagir aussi bien avec le matériel qu'avec les services noyaux du système d'exploitation. Ce type de composant logiciel est utilisé de manière standard pour ajouter de nouvelles fonctionnalités modulaires (pilotes, ...) au sein des systèmes d'exploitation sans accroître, outre mesure, l'empreinte mémoire du noyau.

Nous allons, tout d'abord, décrire l'algorithme de contrôle que nous avons mis en œuvre au sein de notre module, pour, dans les sections suivantes, nous concentrer sur les différentes particularités techniques mises en œuvre dans notre implémentation. Nous allons notamment exposer les configurations réalisées au niveau du matériel afin de mesurer le trafic mémoire pour, ensuite, détailler l'interface de programmation que nous avons utilisée pour surveiller ledit trafic. Nous décrirons ensuite la manière dont nous avons intégré les tables d'estimations de ralentissements au sein de notre module pour, dans une dernière sous partie, développer le mécanisme de suspension des cœurs *best effort* mis en œuvre.

5.3.1 Algorithme

Notre module va lors de son chargement prendre comme données d'entrée, les seuils de ralentissement maximums qui sont tolérés par les applications temps réel et les tables de surcoûts utilisées pour, depuis une bande passante mémoire mesurée, estimer les ralentissements de la tâche temps réel.

Lorsqu'une application temps réel est activée, le système d'exploitation va déclencher le composant régulateur en effectuant un appel système (`ioctl`) au module. Celui-ci va alors configurer les compteurs matériels de mesures du trafic mémoire, et démarrer le mécanisme d'échantillonnage qui va periodiquement appeler une **fonction de contrôle**.

L'algorithme de notre fonction de contrôle s'effectue en trois temps

1. Tout d'abord, dans une étape d'accès aux compteurs, la fonction lit, auprès du matériel, le trafic mémoire réalisé depuis la dernière mesure effectuée et remet à zéro les compteurs.
2. Les valeurs lues sont utilisées pour récupérer, dans la table d'estimation des ralentissements, une évaluation du surcoût temporel subi par la tâche temps réel au sein de l'échantillon courant.
3. Lorsque le cumul du surcoût temporel estimé devient supérieur au seuil fixé par le concepteur du système, le dispositif prend alors la décision de stopper les applications *best effort* exécutées, éliminant ainsi les interférences temporelles pour préserver le respect de l'échéance de l'application temps réel ordonnancée.

   En pratique, nous prenons le seuil fixé par le concepteur auquel nous soustrayons une période temporelle équivalent au pourcentage de temps qu'une application peut passer dans un échantillon, pour éviter le pire cas où une application temps réel n'effectue aucun progrès dans un échantillon dépassant ainsi le seuil fixé.

A la fin de chaque activation de la tâche temps réel, les applications *best effort* éventuellement stoppées sont redémarrées. Le dispositif de contrôle du système effectue, en outre, un *flush* des caches L1 du cœur temps réel. Cette opération permet d'éviter que le cache ne soit plein de données sales qui, lors de la prochaine activation d'une tâche, seront écrites en mémoire principale, perturbant ainsi le comportement de l'application temps réel prochainement activée. En plaçant le flush après l'activation des tâches *best effort*, nous faisons en sorte que le trafic mémoire engendré par une telle opération ne s'ajoute pas au trafic généré par les tâches temps réel.

5.3.2 Mesure de la consommation mémoire

Pour mesurer, de manière expérimentale, la consommation mémoire générée par la pile logicielle, nous pouvions, au choix, utiliser les compteurs matériels des caches L1, du cache L2 ou du contrôleur mémoire. Nous souhaitions mesurer le trafic mémoire source de contention sur les bus c'est à dire le trafic qui se produit au niveau de la hiérarchie mémoire partagée entre tous les cœurs. Les compteurs placés dans les caches L1, effectuent une mesure du trafic mémoire généré vers le cache L2 et, de ce fait, ne répondent pas à nos besoins.

Nous avons donc choisi d'utiliser les compteurs du contrôleur MMDC, décrits précédemment en section 4.1.4, que nous avons réussi à activer sur notre plate-forme et qui nous permettent d'obtenir une mesure de la bande passante générée vers le dernier niveau de la hiérarchie mémoire.

5.3.3 Surveillance par échantillonnage

Idéalement, notre composant régulateur devrait être capable d'estimer le surcoût temporel généré par chaque accès mémoire effectué. Or, le coût en performances d'une telle solution mise en œuvre logiciellement est prohibitif, seule une implantation matérielle au sein du contrôleur mémoire pouvant garantir les performances nécessaires à la mise en œuvre efficace d'une telle approche.

Nous avons donc choisi de surveiller la consommation de bande passante mémoire en utilisant une approche par échantillonnage dans laquelle, régulièrement, le module de contrôle effectue, auprès du contrôleur MMDC, une lecture des accès mémoire réalisés par la pile logicielle.

Pour implanter ce dipositif de mesures périodiques, nous avons utilisé l'interface de programmation **hrtimers** mis à disposition par le noyau Linux [48]. Cette interface a été développée pour gérer les temporisateurs qui se déclenchent à une haute fréquence et qui ont besoin d'une bonne précision. Chaque temporisateur du système est représenté par une instance d'une structure `hrtimer` qui contient une date d'échéance et un pointeur sur une fonction de rappel représentant l'action à effectuer lorsque le temporisateur expire.

La gestion des temporisateurs haute résolution est effectuée pour chaque CPU présent sur la machine, chaque cœur contenant un ensemble de structures `hrtimer` et une horloge matérielle programmable pouvant être configurée pour générer un événement horloge à une date spécifiée sur un cœur donné. La gestion des temporisateurs haute résolution est donc totalement indépendante de l'horloge système utilisée pour incrémenter les `ticks`

Les multiples `hrtimer` armés pour un CPU donné sont stockés dans un **arbre rouge noir** trié selon les dates absolues d'échéance des temporisateurs. L'utilisation d'arbres bicolore permet une insertion et une suppression des nœuds en O(log(N)) ce qui a été considéré par les créateurs comme étant suffisamment efficace pour être utilisé au sein de l'interface. A chaque ajout ou suppression d'un temporisateur, dans l'arbre trié, l'horloge matérielle du cœur est, si nécessaire, réarmée avec la date d'expiration du temporisateur le plus proche.

Lorsque l'horloge matérielle expire, une interruption est alors déclenchée et la routine d'in-

terruption associée va alors supprimer de l'arbre rouge noir tous les temporisateurs expirés appelant au passage la fonction de rappel présente au sein de chacun d'entre eux. Selon la valeur de retour de la fonction de rappel, les temporisateurs sont éventuellement réarmés et réinsérés dans l'arbre entraînant, par la même, une reprogrammation de l'horloge matérielle avec la date d'expiration du temporisateur le plus proche.

L'utilisation de tels temporisateurs a pour avantage de nous permettre de choisir le CPU vers lequel les interruptions sont émises. Nous avons donc configuré notre temporisateur de profilage pour que les interruptions générées soient dirigées sur un des cœurs *best effort* et non sur le cœur temps réel évitant ainsi de perturber notre application critique.

Nous avons également utilisé le sous-système `hrtimers` aussi bien pour tracer les profils mémoire de nos applications dans la partie 5.2.1 que pour générer les traces de consommation mémoire utilisées par le simulateur dans la sous-partie 5.2.4.1, la routine d'interruption alors associée au temporisateur d'échantillonnage sauvegardant en mémoire les mesures effectuées auprès du contrôleur MMDC.

5.3.4 Tables embarquées

Deux principales préoccupations ont guidé l'intégration des tables d'estimations des surcoûts au sein de notre module de contrôle.

Nous avons, tout d'abord, cherché à obtenir les meilleures performances lors de l'opération de recherche des ralentissements estimés depuis une bande passante mesurée. Cette opération, effectuée à une forte fréquence dans la routine de contrôle, ne pouvait être sujette à des temps d'exécution trop longs, sous peine de ne pas être assez réactive pour détecter les interférences, mais aussi, de sur-pénaliser les applications *best effort* exécutées sur le même cœur que la fonction de contrôle. Nous avons donc structuré la table de telle sorte à ce que le surcoût estimé puisse être obtenu en utilisant le trafic mémoire mesuré dans l'échantillon courant comme indice du tableau.

Nous avons, en outre, cherché à minimiser l'empreinte mémoire des tables au sein du régulateur afin, d'une part, de ne pas consommer trop de mémoire mais aussi pour diminuer le trafic mémoire généré par l'opération de recherche, une table de plus petite taille étant plus facilement chargée dans les caches.

Nous avons donc décidé de compresser les données obtenues dans notre table en effectuant une opération de décalage vers la droite sur la valeur lue auprès des compteurs du contrôleur mémoire. Ce décalage se traduit par une diminution de l'amplitude des valeurs mesurées réduisant ainsi la taille du tableau qui est adapté pour prendre en compte un tel décalage.

Le choix de la valeur utilisée dans le décalage fait appel à un compromis entre la nécessité de limiter l'empreinte du tableau en mémoire et l'impact négatif qu'un tel décalage a sur la précision des données contenues dans le tableau. Nous avons choisi d'effectuer un décalage de 10 qui a pour effet secondaire de diviser le nombre d'octets lus par 1024. La valeur utilisée pour l'échantillonnage étant de 50μs, chaque entrée contiguë, dans la table des surcoûts représente un incrément de bande passante de 20,48 MB/s.

Cette approche induit une approximation à deux niveaux différents. Tout d'abord, le ralentissement corrélé à une bande passante observée est arrondi à un multiple de 20.48 , ensuite, nous avons observé qu'en pratique les intervalles échantillonnages n'étaient pas exactement de 50μs. Néanmoins, cette approche permet de minimiser l'impact du mécanisme de contrôle sur les applications *best effort*, lorsqu'elles sont autorisées à s'exécuter en parallèle des applications temps réel maximisant ainsi le parallélisme de notre solution.

Nous avons, pour faciliter l'interfaçage de nos tables avec notre module de contrôle, généré nos tables d'estimations sous la forme de code C intégré au sein d'un co-module qui exporte les symboles des tables vers le module de contrôle.

5.3.5 Suspension des applications

Lorsque le ralentissement estimé de l'application temps réel dépasse le seuil désiré, notre algorithme suspend les applications *best effort*.

Nous avons choisi d'utiliser une **interruption inter-processeurs** nommée aussi *Inter-Processor Interrupt* pour notifier les différents cœurs *best effort* de leur arrêt. Les interruptions inter-processeurs sont un type spécial d'interruption qui peut être généré par le logiciel pour interrompre un autre processeur dans une architecture matérielle multi-cœurs. Le déclenchement d'une telle interruption dans l'architecture ARM est effectué par le logiciel qui écrit dans un registre le numéro de l'interruption qu'il souhaite générer et les identifiants des CPU ciblés par ladite interruption. La demande est alors transmise au *Generic Interrupt Controller* présent au sein du processeur MPCore qui va alors déclencher une interruption vers les cœurs ciblés.

Le système d'exploitation Linux utilisant, par défaut, 5 des 16 interruptions IPI utilisables, nous avons pu modifier le noyau afin d'ajouter une interruption supplémentaire dédiée à la suspension des applications *best effort*. Lorsque la décision de suspension est confirmée, le module de contrôle envoie une interruption à l'ensemble des cœurs qui ordonnancent des applications *best effort*. Sur réception de l'interruption par les cœurs ciblés, chaque application est alors préemptée par le gestionnaire d'interruption qui boucle en attente sur une variable *flag* utilisée pour signaler la fin d'une période d'activation de la tâche temps réel. Lorsque ladite tâche termine sa période d'activation courante, le *flag* est mis à jour entraînant une sortie des gestionnaires d'interruptions et une reprise d'exécution des tâches *best effort*.

L'avantage de cette solution, par rapport à celles qui reposent sur l'utilisation de l'ordonnanceur pour dé-ordonnancer les tâches *best effort*, réside dans sa très faible latence.

5.4 Evaluation

Nous allons, dans cette partie, évaluer l'efficacité de notre solution sur les applications de la suite de test MiBench qui, en présence de contention mémoire, peuvent subir un accroissement de leurs temps d'exécution de plus de 10%. Nous avons choisi de porter notre attention sur les applications dont les temps d'exécutions sont inférieurs à 50ms. En effet, sur de telles applications, la durée d'une période de contrôle représente un pourcentage important du temps total d'exécution induisant un risque important de dépasser le seuil de surcoût voulu.

Nous avons expérimenté notre approche sur la plate-forme logicielle et matérielle décrite dans les sections 4.3 et 4.1 avec une configuration de partage du cache L2 avec 1/4 d'alloué à la tâche temps réel, le reste étant alloué aux tâches *best effort*. Nous avons fixé le seuil de dépassement à 5% du temps d'exécution des tâches temps réel ce qui représente une borne minimale sur la précision de mesures de performances.

Dans la première section de ce chapitre, nous allons évaluer le surcoût à l'exécution que notre mécanisme de contrôle peut avoir sur les applications exécutées. Ensuite, dans une deuxième partie, nous allons effectuer une analyse des performances de notre solution d'un point de vue respect des contraintes temps réel et gain de performances.

5.4.1 Surcoût à l'exécution

La période d'échantillonnage utilisée à l'exécution par notre module de surveillance et de contrôle est un paramètre critique de notre solution. Plus la fréquence d'échantillonnage est forte, plus le système est capable de réagir rapidement lorsque notre tâche temps réel est en situation de dépasser son échéance. En contrepartie, le système d'échantillonnage est basé sur des interruptions horloges qui, lancées à haute fréquences, vont impacter les temps d'exécution des applications *best effort* exécutées sur le cœur qui effectue le contrôle. Il s'agit donc de choisir un juste milieu entre une trop grande précision, qui impacterait trop fortement les tâches *best effort* et reviendrait à sacrifier un cœur pour le module de contrôle à l'exécution et une précision trop faible qui ne permettrait pas de suspendre les tâches *best effort* à temps.

Nous avons donc, pour mesurer le ralentissement induit par le mécanisme d'échantillonnage sur les applicatifs, exécuté les applications sélectionnées au sein de la suite de test MiBench en générant simultanément des interruptions horloges à différentes fréquences sur le cœur qui les ordonnance. Les applications ont été exécutées en isolation sur un seul cœur, les cœurs restant étant désactivés. La colonne « cœur best-effort » de la table 5.1 contient les résultats de nos expériences pour des périodes d'échantillonnage allant de 10μs à 50μs. Lorsque la période d'échantillonnage est de 10μs le ralentissement des applications est au maximum de 47.59%, la majorité des applications étant ralentie de 30%. Nous avons considéré qu'un tel surcoût est trop pénalisant pour les applications *best effort*. Avec une période d'échantillonnage de 50μs le ralentissement plafonne à 16% pour `qsort small` les autres applications oscillant entre 2 et 4%. Nous avons donc sélectionné la période d'échantillonnage de 50μs comme adéquate pour la mise en œuvre de notre mécanisme de contrôle.

Nous pouvons néanmoins constater que, pour les applications temps réel ayant de faibles durées d'exécution, une période d'échantillonnage de 50 µs représente une portion non négligeable de leur temps d'exécution. Pour les applications `susan -c small` et `susan -e small` dont les temps d'exécutions maximaux sont respectivement de 1,15 ms et de 2,13 ms, la durée d'exécution d'un échantillon est au moins égale à, respectivement, 4.3% et 2.3% de la durée de l'application. La mise en œuvre de notre approche, consistant à couper les cœurs *best effort* un échantillon avant que le seuil de 5% ne soit atteint, avec une période d'échantillonnage de 50 µs va se traduire potentiellement, sur de telles applications, par un faible taux de parallélisme. En effet, la présence d'un trafic mémoire, même faiblement intensif, risque d'entraîner une coupure des cœurs *best effort* dès le début de l'exécution des tâches notre approche restant

néanmoins valide pour garantir le respect des contraintes temporelles.

Nous avons également souhaité vérifier que le mécanisme d'échantillonnage exécuté sur le cœur *best effort* ne perturbe pas les applications temps réel exécutées en parallèle. Nous avons donc lancé les applications de la suite MiBench sur le cœur temps réel en parallèle de notre mécanisme d'échantillonnage lancé sur le cœur *best effort* qui exécute une tâche « idle » pour mesurer les variations temporelles induites par les interruptions sur les applications temps réel. Les résultats de ces expériences, sont présentés dans la colonne cœur « temps-réel » du tableau 5.1, avec une période d'échantillonnage qui varie de 10 µs à 50 µs. Nous observons que, quelque soit la période d'échantillonnage utilisée, le ralentissement provoqué par les interruptions déclenchées sur un cœur n'impacte pas significativement les performances des applications ordonnancées sur des cœurs adjacent, le ralentissement observé oscillant entre -0,70% et 0,90%.

	Cœur temps réel		**Cœur best-effort**	
Application	**10 μs**	**50 μs**	**10 μ**	**50 μs**
adpcm -d small	0.26%	-0.50	30.59 %	3.63 %
adpcm -e small	0.06%	-0.27%	31.87 %	4.07 %
fft small	0.59%	0.69%	30.30 %	2.52%
fft -i small	0.90%	0.22%	30.51 %	2.82 %
patricia small	0.12%	0.03%	47.59 %	4.14 %
qsort large	0.44%	0.62%	34.00%	3.19 %
qsort small	-0.12%	-0.70%	32.00%	16.7 %
rijndael -d small	0.26%	0.00%	30.70 %	2.76 %
rijndael -e small	0.67%	0.27%	31.42%	3.19%
sha small	0.14%	-0.33%	29.83 %	2.83 %
susan -c large	0.15%	0.39%	31.60%	2.90%
susan -c small	-0.19%	0.85%	31.80 %	3.53 %
susan -e small	0.45%	0.62%	29.64 %	3.28 %

TABLE 5.1 – Ralentissements des appplications de la suite MiBench provoqué par les interruptions

5.4.2 Efficacité pour des charges constantes

Nous allons, dans cette partie, évaluer sur deux axes la plus-value apportée par notre solution. Tout d'abord, nous souhaitons mesurer le ralentissement généré par les tâches *best effort* sur les applications temps réel pour nous assurer que celui-ci reste en dessous du seuil voulu. Ensuite, nous allons mesurer le gain de performances apporté par notre approche en nous

comparant à la solution « tout ou rien » décrite dans la section 3.2.1.2.

5.4.2.1 Respect des contraintes temps réel

Nous avons, tout d'abord, étudié l'impact de notre solution, sur les applications sélectionnées au sein de la suite de test de MiBench, en les exécutant en parallèle de charges utilisant les mêmes paramètres que nous avons utilisés pour créer les tables de surcoûts. Nous avons, au total pour chaque application, 18 différentes valeurs de bandes passantes générées par les charges, avec 11 différents ratios de lecture-écriture. Nous avons, en outre, fait varier les bandes passantes des différentes charges de telle sorte à ordonnancer sur les cœurs *best effort* des charges ayant des demandes de bande passante asymétriques selon 5 configurations différentes. Au total, nous avons donc effectué 990 expériences par application sélectionnée, chaque expérience impliquant 20 exécutions.

Afin de calculer le ralentissement des applications, nous avons utilisé, comme temps de base pour chaque application, le pire temps d'exécution de l'application exécutée seule avec le cache L2 partitionné. La figure 5.10, contient pour toutes les exécutions des applications en parallèle de charges, les différents ralentissements mesurés sous la forme d'une boite à violon ou chaque boite à violon affiche, pour une application donnée, la distribution des différents ralentissements de toutes ses exécutions. Le surcoût maximal est atteint par l'application `sha small` qui est ralentie de 5.10%, toutes les autres applications ayant un ralentissement inférieur à 5%.

Les valeurs négatives observées pour les applications `susan small -c` et `susan small -e` peuvent être expliquées par le fait que la valeur utilisée comme point de référence est le pire cas observé. Or, notre solution de contrôle empêche l'occurrence du problème d'arriver. Nous expliquons les larges variations observées dans les surcoûts de `susan small -c` et de `susan small -e` par le fait que ces applications ayant une faible durée d'exécution, elles sont extrêmement sensibles à toutes les interférences externes, qu'elles proviennent de la mémoire ou du système d'exploitation.

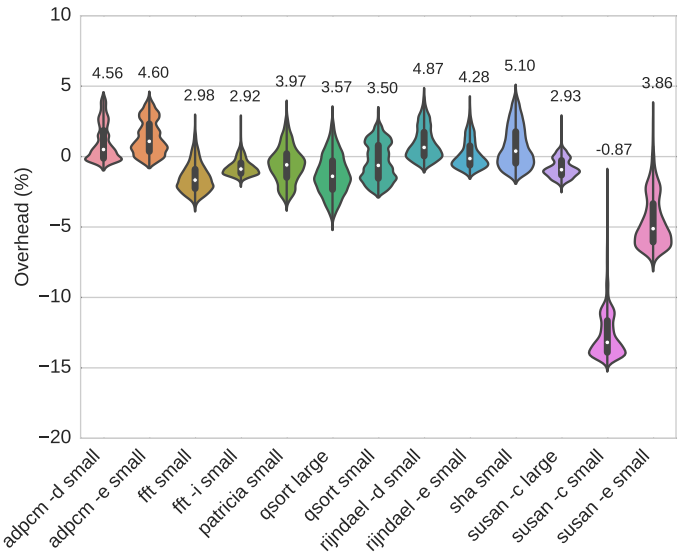


FIGURE 5.10 – Ralentissement des applications sélectionnées au sein de la suite MiBench

5.4.2.2 Parallélisme

Nous avons ensuite choisi de mesurer le gain de performances apporté par notre solution. Nous prenons comme point de comparaison la solution « tout ou rien » développée par SYSGO [41] et présentée dans la partie 3.2.1.2 dans laquelle une tâche critique qui ne doit pas subir d'interférences est exécutée en isolation, les autres cœurs étant désactivés. Nous avons donc défini la métrique de **parallélisme** comme étant le pourcentage de temps durant lequel les applications *best effort* sont exécutées en parallèle de l'application temps réel. Plus le parallélisme est élevé, plus le gain est important, les multiples cœurs du matériel étant plus longtemps utilisés.

Nous avons, pour modéliser la sollicitation des charges sur la hiérarchie mémoire, introduit le terme de **bande passante mémoire demandée**. Cette métrique est calculée, pour un ensemble de charges ordonnancées en parallèle sur notre plate-forme matérielle, en effectuant la somme de la bande passante mémoire mesurée auprès du contrôleur mémoire pour chacune des charges exécutée en isolation. Nous pouvons noter que la métrique de bande passante mémoire demandée est différente de celles obtenues par les charges qui vont, lorsqu'elles sont exécutées sur la plate-forme matérielle, générer de la contention mémoire qui va les ralentir les unes par rapport aux autres ce qui va, in-fine, diminuer leur consommation de bande passante. La bande passante demandée peut être vue comme la bande passante qui serait mesurée si les problèmes de contention mémoire n'existaient pas sur notre plate-forme.

La figure 5.14 montre le degré moyen de parallélisme atteint pour chaque application temps réel exécutée en parallèle des multiples et diverses charges qui varient tant en termes de bande passante demandée que de taux de lecture écriture. Nous avons tracé, pour chaque application, cinq sous-figures différentes. A chaque sous-figure correspond une combinaison de charges bien précise qui varie tant en terme du nombre de charges activées, de une à trois charges,

que de symétrie des bandes passantes générées par les charges. Dans la configuration deux charges asymétriques, une des charges génère environ 35 pourcent de la bande passante demandée tandis que dans la configuration trois charges asymétriques une des charges produit 50 pourcent du totale de la bande passante demandée, les deux charges restantes engendrant respectivement 30 et 20 pourcent de la bande passante demandée restante. Nous avons, pour chaque sous-figure, affiché les taux de parallélisme atteint en différenciant les ratios de lecture et d'écriture utilisés au sein des charges.

Tout d'abord, nous pouvons observer que les variations de la configuration des charges tant en termes de symétrie que du taux de lecture/écriture n'ont pas un grand impact sur le taux de parallélisme obtenu.

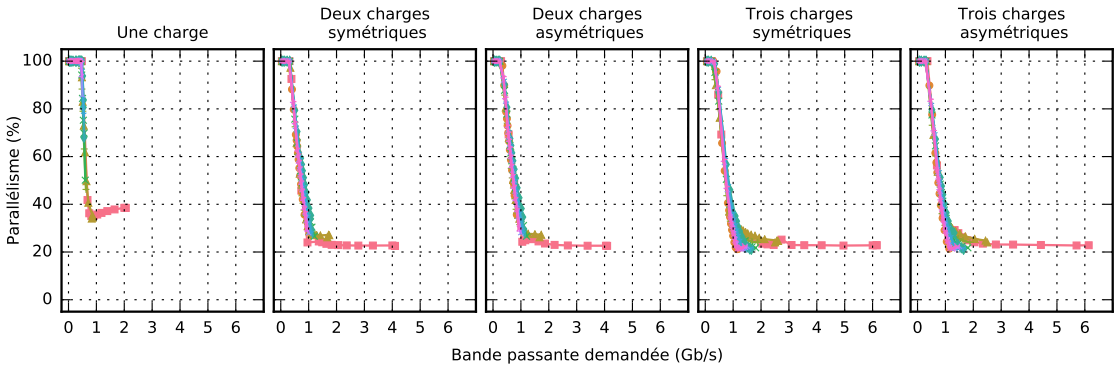
Pour 7 des 13 applications (`adpcm -e small`, `adpcm -d small`, `fft small`, `fft -i small`, `rijndael -d small`, `rijndael -e small`, `sha small`), nous atteignons, dans toutes les configurations de charges, au moins 70 % de parallélisme dès lors que les charges génèrent moins de consommation mémoire. La limite de bande passante demandée en dessous de laquelle les 7 applications obtiennent, dans toutes les configurations de charge, 70% de parallélisme varie grandement selon les applications allant de 532 Mb/s pour `rijndael -e small` à 6148 Mb/s pour `fft small`.

Le degré de parallélisme de `patricia small` dépend du nombre de cœurs ordonnançant des charges. Si le parallélisme peut atteindre 100% lorsqu'un ou deux cœurs ordonnancent des charges, il ne dépasse pas les 70% avec une configuration de trois charges asymétriques et plafonne à 55% avec trois charges symétriques ordonnancées.

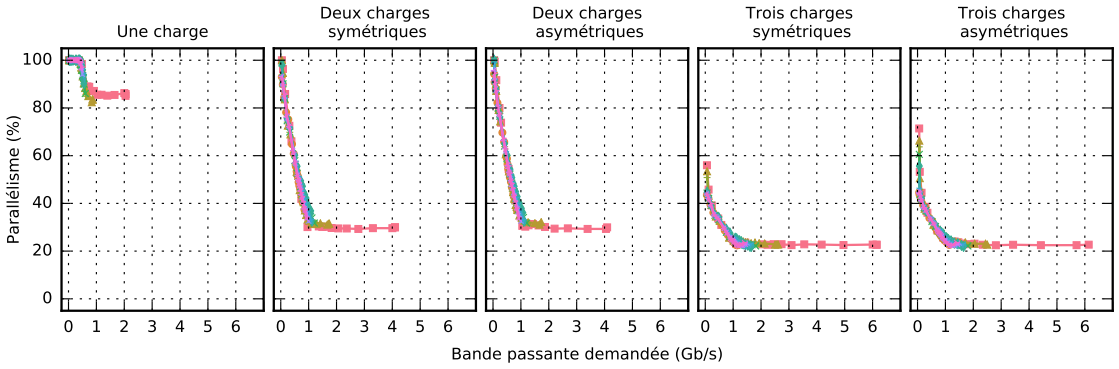
La même observation peut être faite pour `qsort small` où le parallélisme maximal atteint les 100% lorsqu'une charge est ordonnancée pour descendre aux alentours de 67% lorsque deux charges sont exécutée (66% en configuration symétrique et 69% en configuration asymétrique) et plafonner à 44% lorsque les trois charges sont activées.

Le taux de parallélisme maximal de `susan large -c` atteint quasiment 100% (10%, 95% et 98%) pour les configurations avec une et deux charges d'ordonnancées et ce, quel que soit les symétries et s'effondre en dessous de 8% lorsque trois charges sont activées. Enfin, si `susan small -e` peine à atteindre 60% de parallélisme avec une seule charge d'activée, son taux s'effondre aux alentours de 10% dès lors que le deuxième et troisième cœur sont activés.

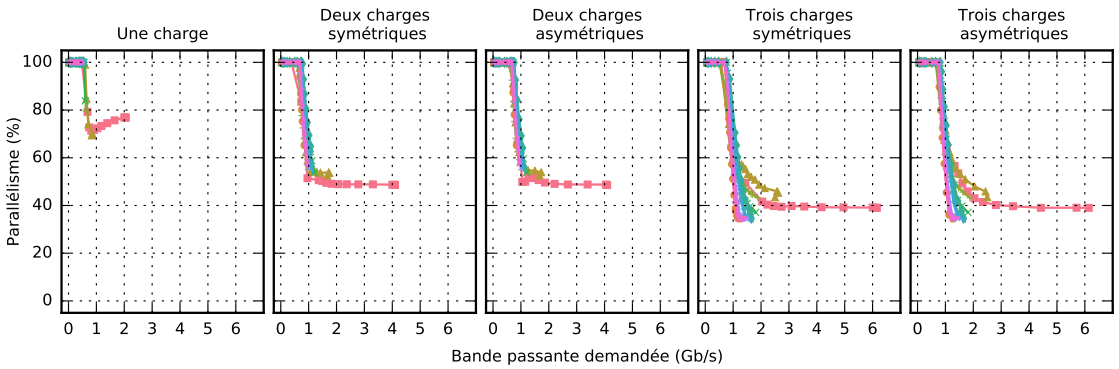
Enfin, pour `qsort large` nous observons peu de différences en termes de parallélisme entre les différentes configurations de charges. Le parallélisme maximal atteint est aux alentours de 30% pour les cinq configurations de charges testées et s'effondre rapidement dès lors que la bande passante augmente. Enfin, nous notons que `susan small` ne parvient pas à obtenir le moindre degré de parallélisme.



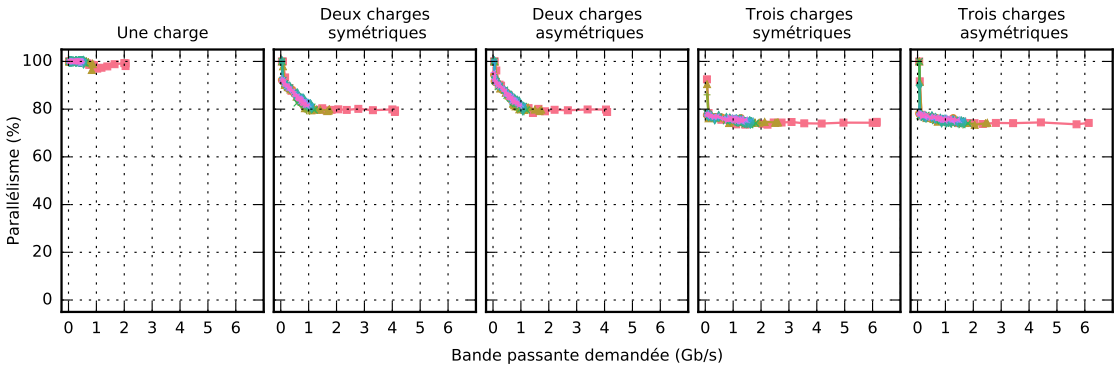
(a) Adpcm -e small



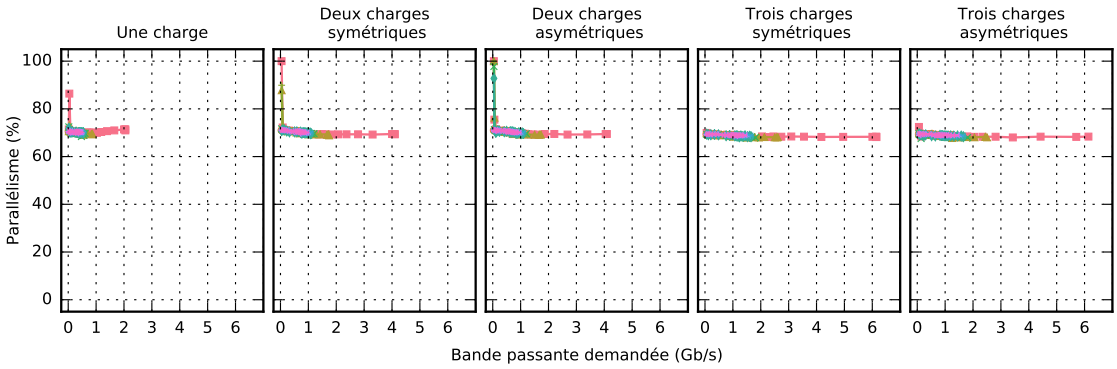
(b) Patricia small



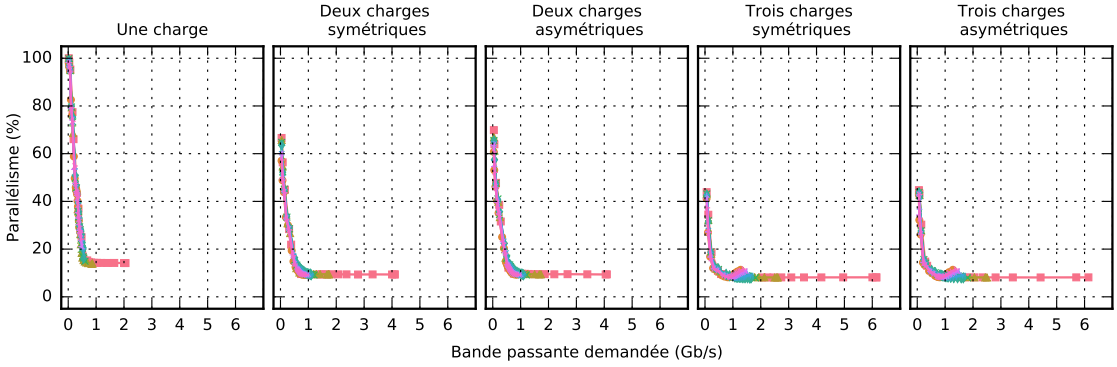
(c) Adpcm -d small



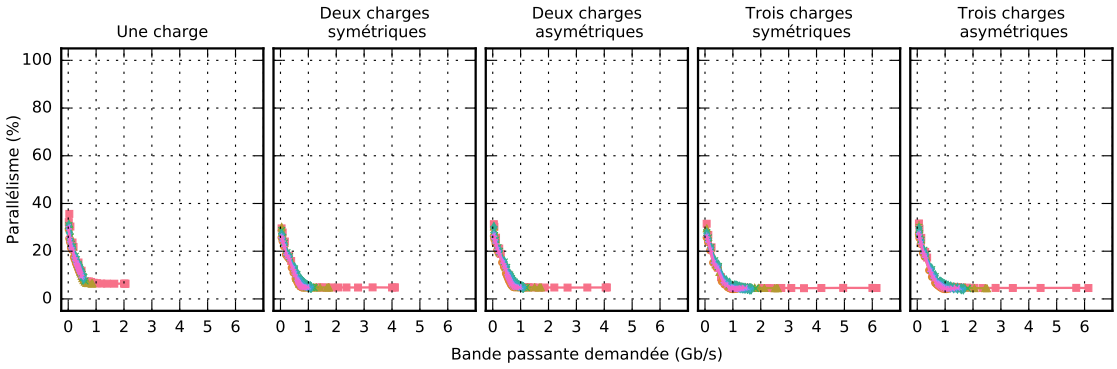
(d) Fft small



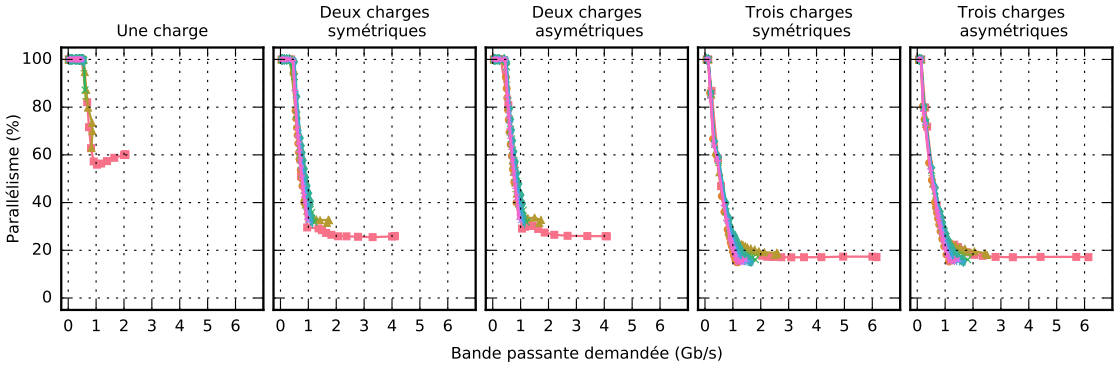
(a) Fft -i small



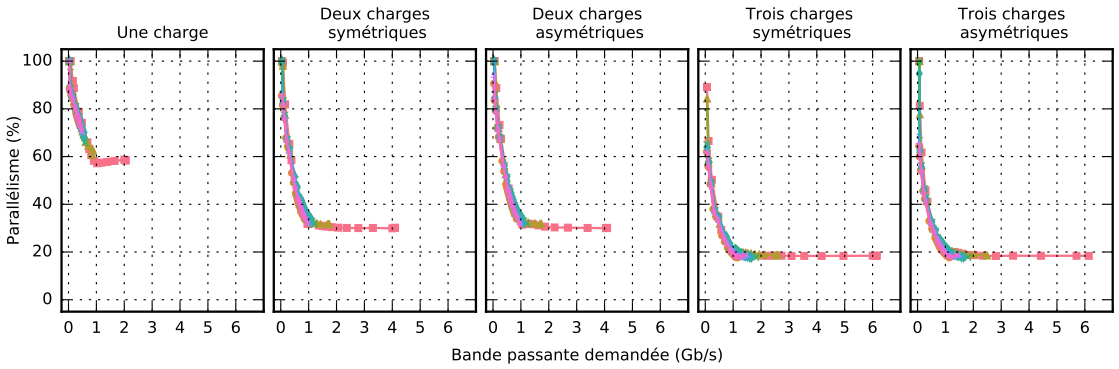
(b) Qsort small



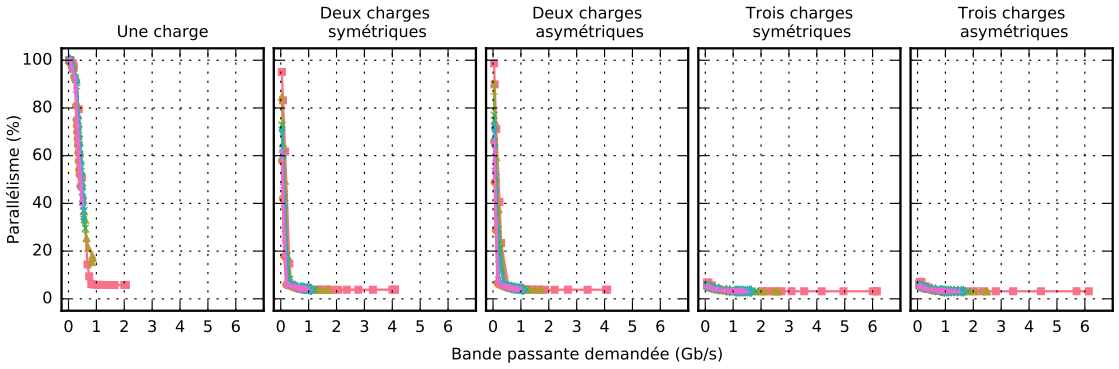
(c) Qsort large



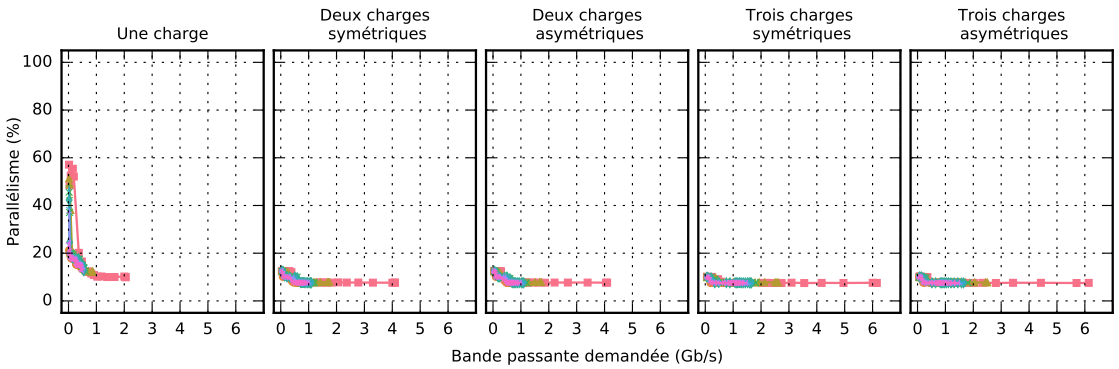
(d) Rijndael -d small



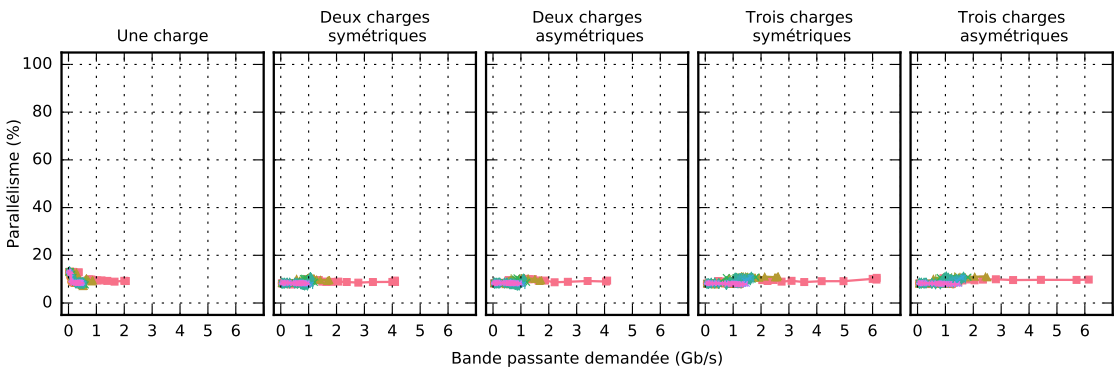
(a) Rijndael -e small



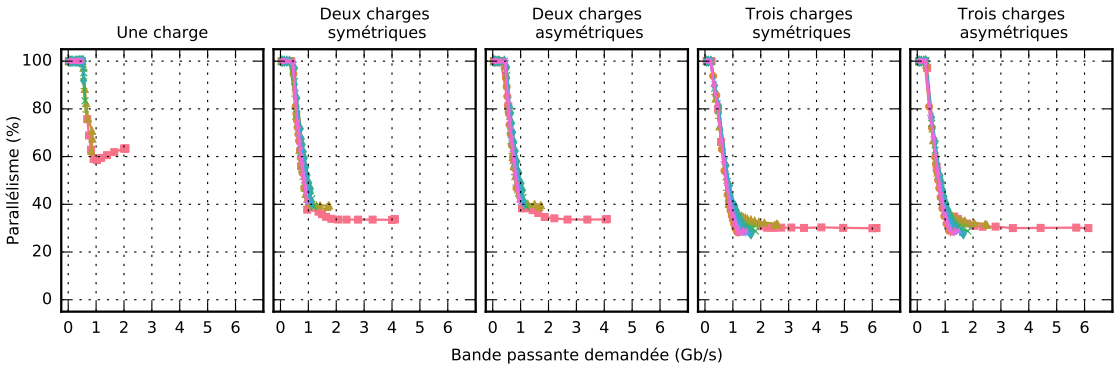
(b) Susan -c large



(c) Susan -e small



(d) Susan -c small



(a) Sha small

FIGURE 5.14 – Parallélisme

5.5 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche de régulation pour exécuter sur une même carte multi-cœurs COTS, des applications temps réel en parallèle d'applications *best effort*, le tout en bornant les ralentissements provoqués par les applications du deuxième type sur les applications du premier type.

Notre approche se déroule en deux étapes : dans une première étape hors ligne exécutée une seule fois par application, nous effectuons une caractérisation des impacts de la contention mémoire pour une plate-forme logicielle et matérielle donnée. Le résultat de cette étape est un oracle capable, depuis une bande passante mesurée globalement et un nombre de cœurs *best effort* actifs donné, d'estimer les ralentissements qui peuvent impacter une tâche temps réel.

Dans une étape en ligne, effectuée systématiquement sur les systèmes en production, nous exécutons en parallèle les applications temps réel et *best-effort*. Nous activons préalablement un composant régulateur qui, à partir de la bande passante globale mesurée, va détecter les ralentissements impactant les tâches temps réel et, s'ils deviennent trop importants, suspendre les applications *best effort* le temps que l'application temps réel termine son activation.

Nous avons investigué la pertinence de notre approche sur 13 applications de la suite *MiBench*, qui, en présence de contention mémoire, peuvent subir un ralentissement de plus de

10%. Sur ces 13 applications, 12 d'entre elles sont ralenties de moins de 5% lorsque notre mécanisme est activé, une application atteignant les 5.10% de ralentissement. Nous obtenons pour 7 des applications dans toutes les configurations de charges, au moins 70% de parallélisme dès lors que les charges génèrent peu de consommation mémoire.

Dans le chapitre suivant, nous allons effectuer une étude des codes sources des applications de la suite *Mibench* pour déterminer les causes des pics de consommation mémoire.

Chapitre 6

Analyse des causes de la sensibilité mémoire des applications temps réel

Sommaire

6.1 Profilage des applications . . . 130
6.1.1 Profileur mémoire . . . 130
6.1.2 Classification . . . 134
6.1.3 Méthodologie d'investigation des pics de bande passante mémoire . . . 138
6.2 Impacts des entrée/sortie . . . 139
6.2.1 Analyse du problème . . . 139
6.2.2 Conclusion & solutions . . . 144
6.3 Impacts du système d'exploitation . . . 145
6.3.1 Analyse du problème . . . 145
6.3.2 Conclusion & solutions . . . 146
6.4 Impacts des algorithmes applicatifs . . . 146
6.4.1 Analyse du problème . . . 146
6.4.2 Conclusion & solutions . . . 148
6.5 Conclusion . . . 148

Nous allons, dans ce chapitre, effectuer une étude de la consommation mémoire des applications temps réel de la suite de test *Mibench*. Notre but est de confronter les profils mémoire applicatifs présentés dans le chapitre précédent 5.4 au code source des applications pour comprendre les raisons qui rendent une application sensible à la contention.

Tout d'abord, dans la première section de ce chapitre, nous listerons les différentes limitations du profileur, que nous avons utilisé auparavant dans la section 5.3.3, et qui nous ont amené à développer et à tester un nouvel outil.

Puis, nous utiliserons ce nouveau profileur pour effectuer une classification de nos applications en deux catégories : les applications dont le profil mémoire contient des pics de bande passante et celles qui ont un profil plus lissé.

Ensuite, dans les parties suivantes, nous détaillerons chacune des sources de pics de bande passante que nous avons identifiées et nous proposerons diverses solutions pour y remédier.

6.1 Profilage des applications

Nous allons, dans cette section, présenter les choix de conceptions ayant guidé le développement d'un nouveau profileur mémoire. Nous effectuerons également une étude d'impact pour évaluer l'effet de sonde de notre outil sur les applications profilées.

Ensuite, dans une deuxième partie, nous tracerons les profils des applications de la suite *Mibench* afin d'effectuer une classification en deux catégories : les applications dont le profil mémoire contient des pics de bande passante et celles qui ont un profil plus lissé. Nous détaillerons également la méthodologie que nous avons utilisée pour isoler les causes racines des pics de consommation mémoire.

6.1.1 Profileur mémoire

Si la solution de profilage, développée précédemment pour notre mécanisme de contrôle dans la section 5.3.3, était suffisante pour détecter les phases applicatives générées par l'exécution de blocs de code, elle n'est pas assez précise pour évaluer exactement la consommation mémoire à la granularité d'une instruction. Nous avons donc effectué une évaluation des deux alternatives dont nous disposions pour tracer avec plus de précision les profils de consommation mémoire de nos applicatifs.

La première reposait sur le développement d'un modèle précis de notre plate-forme matérielle comportant notamment le système mémoire et processeur pour pouvoir mesurer, par simulation, la consommation mémoire générée par la pile logicielle. La deuxième piste, expérimentale, consistait à étendre le profileur, que nous avons développé précédemment dans notre solution de contrôle sous la forme d'un module noyau, afin d'utiliser les compteurs de performances de notre matériel pour, par de multiples expériences, mesurer la consommation mémoire générée par la pile logicielle.

Les avantages de la première solution résident dans la possibilité de mettre en œuvre un profilage à grain fin permettant ainsi de caractériser le trafic mémoire généré par chaque instruction exécutée. En contrepartie, cette solution impliquait la création d'un modèle précis de la plate-forme matérielle ce qui est, par nature, complexe. Sa création, en outre, nécessite la présence de spécifications du matériel plus détaillées que celles fournies par les fondeurs pour valider la représentativité du modèle. La deuxième approche, que nous avons choisie, ne possède pas ces inconvénients mais repose sur la confiance que l'on peut accorder aux compteurs du matériel.

La section suivante s'attache à détailler le fonctionnement de ce nouveau mécanisme.

6.1.1.1 Choix de conception

Un profil mémoire applicatif idéal devrait contenir une bijection entre chaque instruction de l'application exécutée sur le processeur et la consommation mémoire générée par ladite instruction. La mise en œuvre d'une telle approche, dans une démarche expérimentale, ne peut être réalisée que par une implémentation matérielle. Il s'agit, en effet, de taguer les requêtes mémoire générées par chaque instruction exécutée qui se propagent à travers la hiérarchie des caches en générant de nouvelles requêtes (lignes de caches sales évincées, ...), jusqu'au contrôleur MMDC chargé d'enregistrer le profil ainsi obtenu. Cette fonctionnalité n'étant pas présente sur notre plate-forme, nous avons donc choisi d'utiliser une approche logicielle par échantillonnage pour mesurer, à intervalles réguliers, la consommation mémoire générée par la plate-forme logicielle.

Dans une telle approche, la période d'échantillonnage utilisée par le profileur revêt une importance capitale. En effet, chaque échantillon mesuré représente une moyenne de la consommation mémoire effectuée entre deux mesures. Le choix d'une période d'échantillonnage trop grande peut donc engendrer un effet de lissage sur le profil mémoire occasionnant une suppression ou une diminution des pics de consommation mémoire. La capacité à mettre en œuvre un intervalle d'échantillonnage suffisamment petit pour obtenir une mesure précise est donc essentielle.

La solution usuelle pour effectuer un échantillonnage réside dans l'utilisation de temporisateurs tel que proposé par l'interface de programmation **hrtimers** du noyau Linux que nous avons utilisée dans la partie 5.3.3 du chapitre précédent. Cependant, Peter et al. [103], ont démontré que des temporisateurs réglés avec des intervalles de temporisation trop petit, n'arrivent pas à respecter le cadencement imposé. Afin de pouvoir effectuer un profilage, avec une résolution descendant jusqu'à 1 µs, nous avons donc choisi d'utiliser une technique d'échantillonnage reposant sur une boucle d'attente active exécutée sur un cœur dédié.

Nous avons mis en place un processus de profilage en plusieurs étapes :

1. **Initialisation**
   Dans la première étape d'initialisation, effectuée avant le démarrage de l'application, le module alloue en mémoire vive un tampon de la taille ad hoc destiné à recueillir les valeurs lues auprès des compteurs matériels. Un *thread* noyau est alors créé et verrouillé sur un cœur libre de toute application avec une politique d'ordonnancement *FIFO* et une priorité maximale. Il va, dans sa routine d'exécution, effectuer les étapes de configuration et d'initialisation des compteurs de mesure du trafic mémoire présents au sein du contrôleur MMDC et du compteur de cycles du processeur, sur lequel le *thread* est ordonnancé pour, ensuite, se bloquer en attente sur une barrière de synchronisation.
2. **Exécution**
   Lorsque l'application commence son exécution, le *thread* de profilage est débloqué de sa barrière et démarre alors les compteurs matériels. Il collecte ensuite de manière cyclique les différentes valeurs auprès des multiples compteurs activés, chaque valeur mesurée étant regroupée au sein d'un unique échantillon contenant le nombre d'octets lus, le nombre d'octets écrits et la valeur du compteur de cycles du processeur. Après chaque

opération de collecte, le *thread* effectue un appel à la fonction `ndelay(unsigned long nsecs)` du noyau Linux qui effectue une attente active durant `nsecs` nanosecondes. En paramétrant la valeur passée à la fonction `ndelay` nous pouvons faire varier le cadencement de l'échantillonnage. Chaque échantillon collecté est sauvegardé en mémoire vive dans le tampon préalablement alloué.

3. **Terminaison**
   Une fois que l'application profilée a terminé son exécution, les différents compteurs matériels sont stoppés et les multiples échantillons collectés en mémoire vive sont sauvegardés dans une mémoire de stockage de masse. La bande passante mémoire des applications est calculée hors ligne en utilisant, pour chaque échantillon collecté, le nombre d'octets lus et écrits et la durée d'échantillonnage. La boucle d'attente active utilisée dans notre approche produisant de petites variations temporelles entre chaque échantillon, nous utilisons la valeur du compteur de cycles présent dans chaque échantillon pour effectuer un réajustement temporel, le processeur étant cadencé à 1 GHz, un cycle du CPU s'exécute en une nanoseconde.

Pour éviter que le profileur n'interfère avec les applications, nous avons appliqué les dispositions déjà mises en œuvre dans la section 4.3.2. Les applications profilées ont été verrouillées sur un seul cœur (Le cœur 0), le profileur étant verrouillé sur un autre cœur (Le cœur 1), les cœurs restants étant désactivés. Afin d'éviter toutes préemptions intempestives, les applications observées et le *thread* de profilage sont ordonnancés en utilisant la politique `SCHED_FIFO` avec la plus forte priorité. Nous avons également désactivé le dispositif dit de *Real Time throttling* qui redonne le contrôle au système d'exploitation si une tâche est ordonnancée pendant une durée trop longue. Enfin, nous avons réduit les interférences mémoire entre les applications profilées et le profileur en partitionnant le cache L2 entre les différents cœurs.

Les avantages de cette approche résident dans la possibilité d'utiliser un intervalle de profilage le plus faible possible. En contrepartie, il est nécessaire de dédier un cœur au profileur ce qui, dans notre cas d'utilisation, n'est pas problématique.

6.1.1.2 Étude d'impact

Le choix de l'intervalle d'échantillonnage, utilisé par le profileur pour effectuer les mesures, peut avoir un impact significatif sur le résultat et la précision du profilage. L'utilisation d'un long intervalle a pour effet de lisser la consommation mémoire, les valeurs profilées représentant une moyenne sur la période d'échantillonnage, ce qui entraîne une réduction de l'amplitude des pics de consommation mémoire dont la durée est plus courte que celle de l'intervalle. L'utilisation d'un intervalle d'échantillonnage plus court permet d'obtenir des informations plus précises concernant les pics de consommation mais peut également provoquer une distorsion du profil mesuré. En effet, la mesure de la mémoire est effectuée globalement et prend en compte l'ensemble des consommations effectuées par les composants matériels de la plate-forme. Le profileur utilisant de la mémoire pour enregistrer les valeurs lues à chaque échantillon, est donc à l'origine d'une partie du trafic mesuré. De plus, ce trafic supplémentaire peut potentiellement retarder les accès effectués par l'application profilée et ainsi, par effet de bord, accroître son temps d'exécution.

Nous avons donc étudié l'impact de la consommation mémoire générée par le profileur sur les applications profilées. A cet effet, nous avons développé une application à faible empreinte mémoire qui incrémente un compteur allant de zéro à une borne maximale. En faisant varier la borne supérieure de notre application, nous pouvons contrôler le temps d'exécution du programme. La consommation mémoire générée par une telle application est faible et se concentre principalement au début de l'exécution lors du chargement du code dans les caches de niveaux un et deux, le trafic mémoire généré en continu étant inexistant. Le profilage de cette application nous permet donc d'isoler la bande passante mémoire générée par le profileur de celle applicative.

La figure 6.1 présente la bande passante obtenue lors du profilage de notre application à faible empreinte mémoire, avec des durées d'exécutions applicatives variant de 1 ms à 50 ms, soit une variation correspondant au temps d'exécution de nos applications temps réel de tests, et avec des temps d'échantillonnage allant de 50 µs à 1 µs ce qui permet, en émettant l'hypothèse optimiste que le processeur est capable d'exécuter une instruction par cycle, de capturer le trafic mémoire toutes 1000 instructions exécutées.

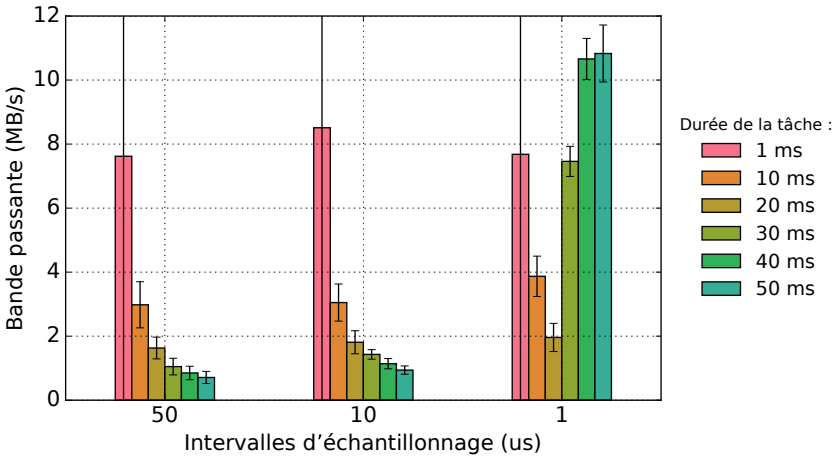


Figure 6.1 – Bande passante mémoire générée par le profileur

Nous observons, un comportement mémoire quasiment identique entre les applications profilées avec une résolution de 50 µs et celles profilées avec une résolution de 10 µs. Les applications dont le temps d'exécution est de 1 ms ont une bande passante qui varie autour des 8 MB/s, avec un important écart type, pour au fur et à mesure que les temps d'exécution des applications augmentent, assister à une diminution de la bande passante mesurée qui décroît aux alentours des 3,5 MB/s, pour un temps d'exécution de 10 ms jusqu'à descendre en dessous de 1 MB/s pour les temps d'exécution les plus grands. Ce comportement peut être expliqué par le coût en bande passante mémoire que représente le chargement de l'application dans les caches. Plus la durée de l'application est longue, plus le coût fixe de chargement de l'application est dilué dans le temps d'exécution qui ne génère aucun trafic mémoire.

L'importante variation des temps d'exécution, pour les applications ayant une durée d'1 ms, s'explique par les perturbations du système d'exploitation décrites ultérieurement dans la section 6.3, la présence de pics mémoire ayant lieu toutes les 10 ms, une exécution sur 10 se voit

donc imputer un surcroît de bande passante.

Nous pouvons également remarquer que la comparaison deux à deux des applications, ayant une même durée d'exécution, pour les intervalles d'échantillonnages 50 µs et 10 µs, montre que les bandes passantes mesurées pour l'intervalle d'échantillonnage 10 µs sont légèrement supérieures à celles mesurées avec l'intervalle d'échantillonnage de 50 µs reflétant ainsi la consommation mémoire supérieure nécessaire à la sauvegarde des échantillons supplémentaires.

Lorsque la fréquence d'échantillonnage de 1 µs est utilisée, l'application d'une durée de 1 ms a un trafic mémoire quasiment égal à celui mesuré pour des fréquences d'échantillonnage plus grande, l'impact du plus grand nombre d'échantillon généré par la fréquence d'échantillonnage plus intensive n'étant pas visible eut égard à la faible durée de l'application.

Quand le temps d'exécution des applications augmente jusqu'à atteindre 20 ms, la bande passante mesurée décroît, le coût de chargement de l'application dans les caches étant amorti par la durée plus grande de l'application. Nous pouvons néanmoins remarquer que la bande passante mesurée pour les applications 10 ms et 20 ms profilées en 1 µs est supérieure à celle mesurée avec un profilage de 10 µs ce qui traduit la consommation mémoire accrue nécessaire à la sauvegarde des échantillons additionnels.

En revanche, le comportement des bandes passantes mesurées pour des temps d'exécution de 30 ms, 40 ms et 50 ms diverge fondamentalement de celui observé pour des résolutions de profilages plus importantes. En effet, l'accroissement de la taille mémoire nécessaire pour sauvegarder les échantillons surnuméraires entraîne une augmentation de la consommation mémoire qui prend le dessus sur l'effet de lissage du surcoût de chargement de l'application dans les caches observé précédemment. La bande passante maximale atteinte est alors de 11 MB/s pour une application dont la durée d'exécution est de 50 ms.

Pour finir, nous avons estimé que les perturbations induites par la bande passante de 11 MB/s générée par le profileur restaient en dessous d'un seuil tolérable pour nos travaux. Enfin, nous n'avons pas observés de différences significatives dans les temps d'exécution des applications profilées avec différentes périodes d'échantillonnage.

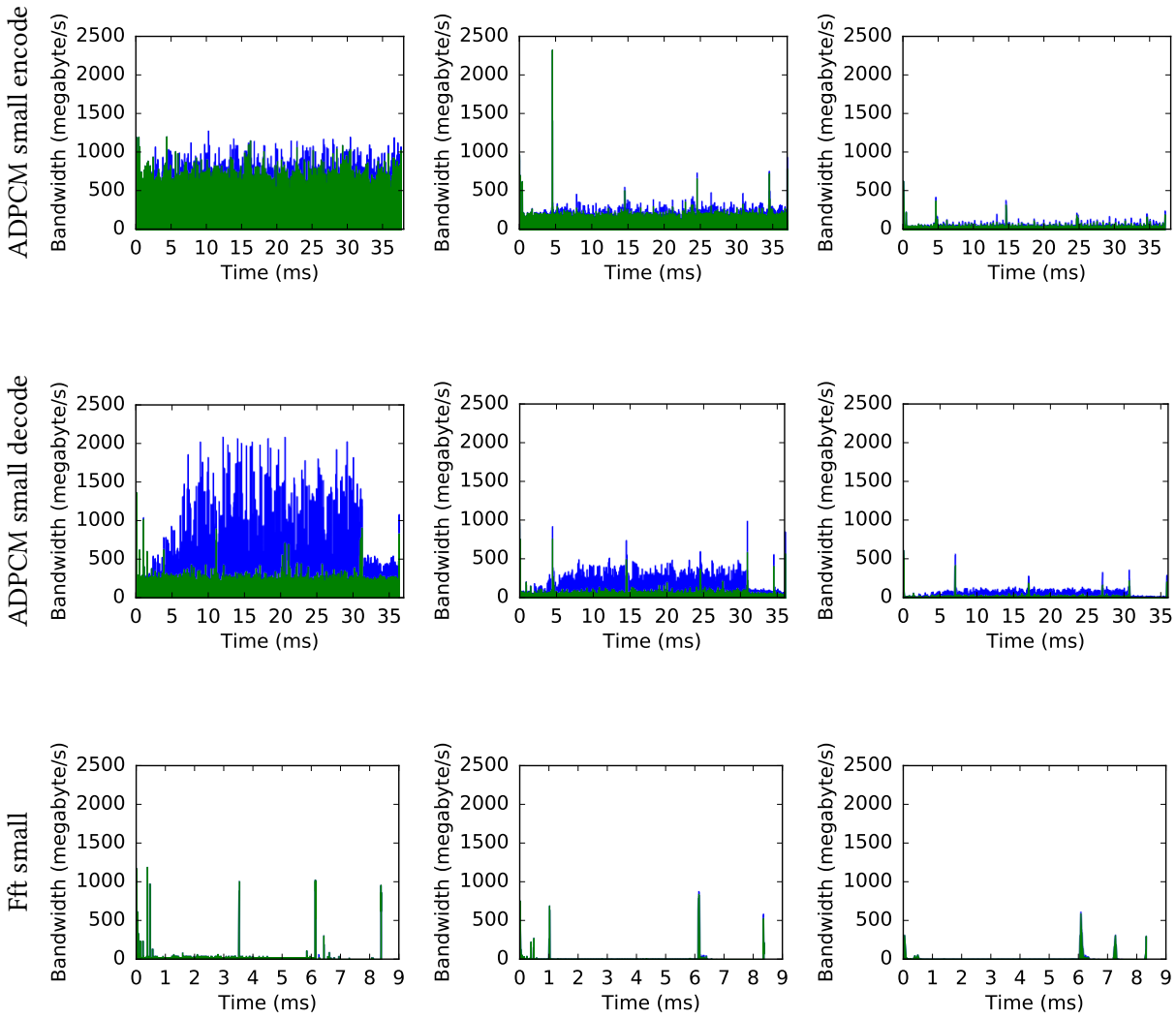
Dans la section suivante, nous allons réaliser une étude des profils applicatifs tracés à l'aide de notre profileur pour de multiples résolutions d'échantillonnage.

6.1.2 Classification

La figure 6.2 contient les profils mémoires des applications MiBench profilées pour différentes périodes d'échantillonnage. Une première analyse de ces profils fait apparaître deux types d'applications :

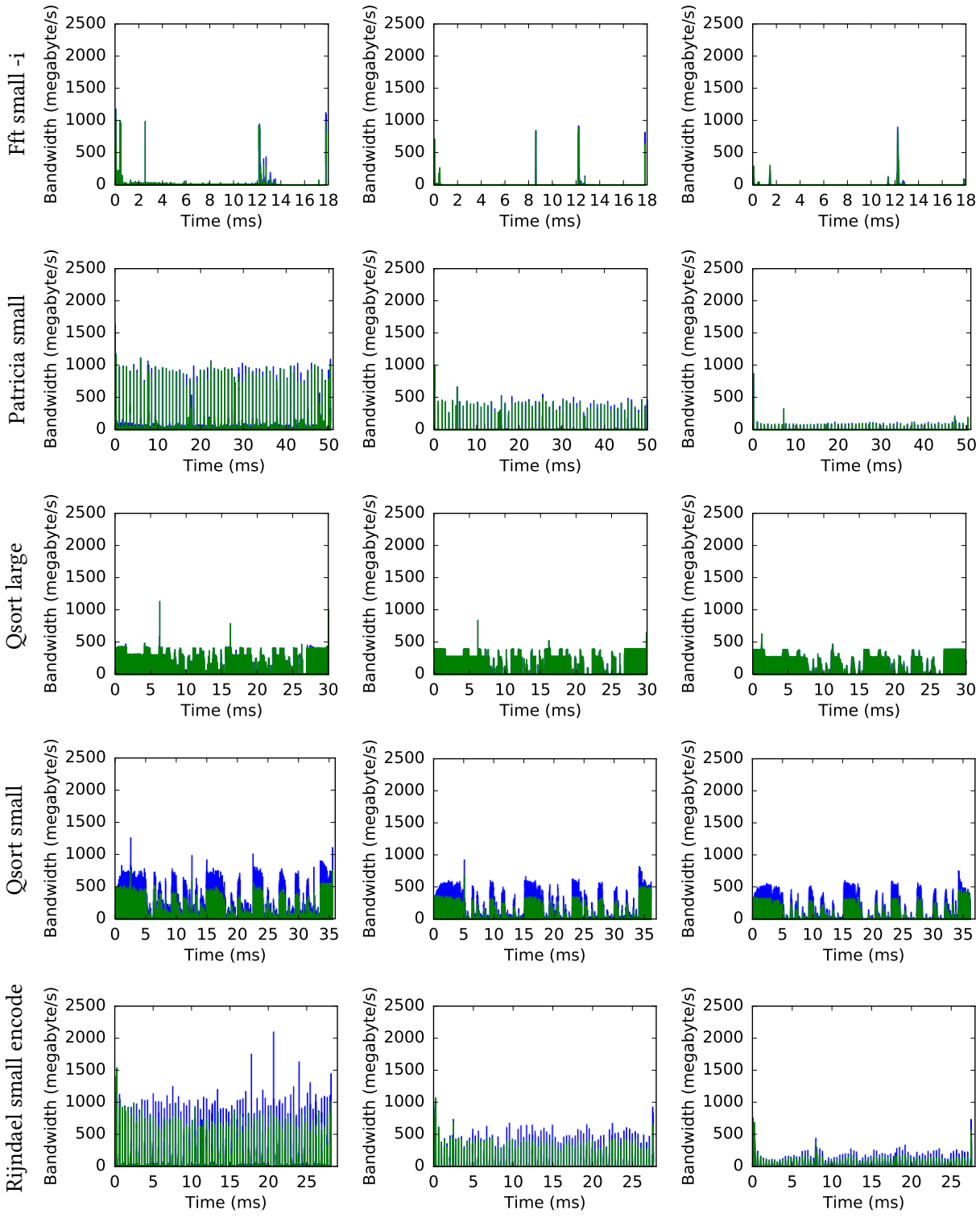
1. Les applications dont les profils contiennent des pics de bande passante réguliers : `ADPCM small encode`, `ADPCM small decode`, `Patricia small`, `Rijndael small decode`, `Rijndael small encode`, `Sha small` et `Susan large -c`
2. Les autres applications qui ont un profil plus lissé : `Fft small`, `Fft small -i`, `Qsort large`, `Qsort small`, `Susan small -c`, `Susan small -e`

Nous constatons que pour les applications du premier type, la diminution de l'intervalle d'échantillonnage de 10 µs à 1 µs et de 50 µs à 10 µs entraîne un accroissement de la hauteur des pics par deux ou plus. Sur de telles applications, nous ne pouvons connaître la bande passante maximale atteinte, les pics de bande passante mémoire observés s'accroissant au fur et à mesure que l'intervalle d'échantillonnage décroît sans jamais se stabiliser. Seule l'utilisation d'un profileur à haute résolution permet de mettre en évidence ces pics de consommation instantanés qui, autrement, sont lissés dans le profil mémoire. En revanche, nous observons que pour les applications du deuxième type, les changements de résolution du profilage n'impactent que modérément le profil obtenu.



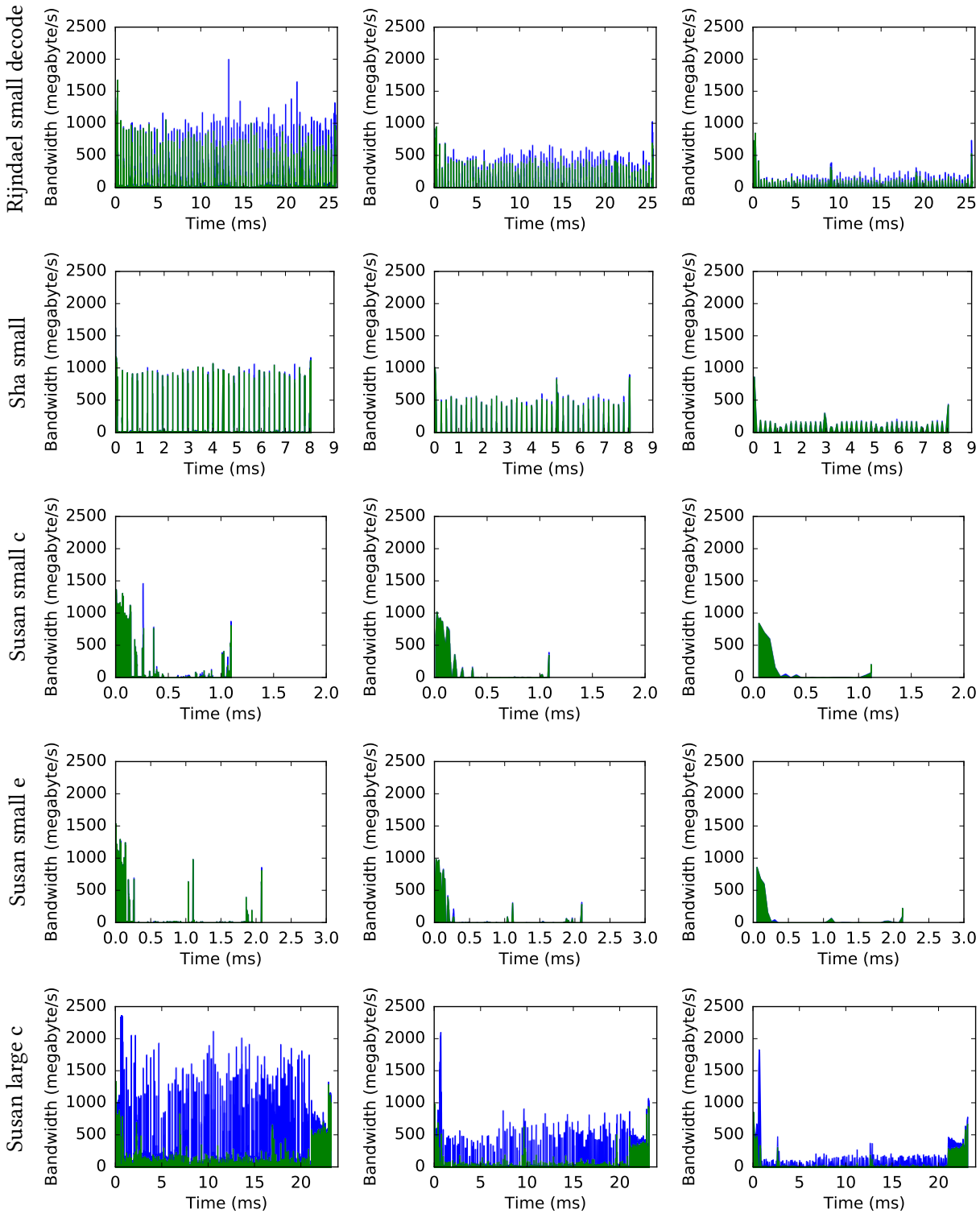
(a) 1 µs (b) 10 us (c) 50 µs

FIGURE 6.2 – Profils mémoire des applications de MiBench exécutées avec différents intervalles d'échantillonnage.



(d) 1 µs (e) 10 us (f) 50 µs

FIGURE 6.2 – Profils mémoire des applications de MiBench exécutées avec différents intervalles d'échantillonnage.



(g) 1 µs (h) 10 us (i) 50 µs

FIGURE 6.2 – Profils mémoire des applications de MiBench exécutées avec différents intervalles d'échantillonnage.

6.1.3 Méthodologie d'investigation des pics de bande passante mémoire

La présence de pics de consommation mémoire au sein d'une partie des applications de la suite de test *Mibench* complique l'analyse des profils applicatifs, le profileur mémoire étant incapable de capturer la véritable distribution de la bande passante générée par les applications.

Afin d'identifier les instructions à l'origine des pics de consommation mémoire, nous avons développé un mécanisme de **tags** permettant d'établir un lien entre des instructions du code source applicatif et le profil correspondant. Nous avons ajouté manuellement, au sein du code source des applications, des instructions afin de lire et d'enregistrer en mémoire la valeur du compteur de cycle du cœur sur lequel l'application est exécutée.

Le profilage d'une application ainsi modifiée suit un processus en trois étapes :

Initialisation
: Lorsqu'une application taguée est profilée, le module noyau va, dans l'étape d'initialisation des compteurs matériels, configurer et activer le compteur de cycles du cœur sur lequel l'application est ordonnancée pour qu'il puisse être lu depuis du code exécuté en mode utilisateur.

Exécution
: Quand le profilage commence, le *thread* noyau du profileur va, en même temps qu'il active les compteurs matériels du cœur sur lequel il s'exécute, démarrer le compteur de cycles du cœur applicatif. L'application va alors commencer son exécution, la valeur du compteur de cycles étant enregistrée en mémoire vive à chaque fois qu'un tag est exécuté.

Terminaison
: Une fois que l'application profilée a terminé son exécution, les différents compteurs matériels sont stoppés, les tags et le profil mémoire contenu en mémoire vive sont sauvegardés dans une mémoire de stockage de masse. Il est alors possible de tracer le profil mémoire applicatif en utilisant les échantillons enregistrés par le profileur puis d'y superposer les occurrences des tags qui ont été enregistrés par l'application en utilisant les valeurs du compteurs de cycles présents à la fois dans les tags et dans les échantillons de profilage pour faire correspondre à chaque tag un échantillon du profil mémoire.

La figure 6.3 affiche un gros plan effectué sur le profil mémoire de l'application `Patricia small` avec un tag posé avant chaque appel à la fonction `fgets`, les différentes rayures verticales représentant l'exécution d'un tag. L'utilisation d'un grand nombre de tags, pouvant entraîner une augmentation substantielle de la durée de l'application taguée, nous avons utilisé le mécanisme de tags uniquement pour identifier les pics de bande passante.

En utilisant ce mécanisme, nous avons pu effectuer une classification des pics mémoire applicatifs en trois groupes : les pics issus de **fonctions d'entrée/sortie**, les pics générés par le **système d'exploitation** et enfin ceux qui proviennent du **code source applicatif**.

Dans les sections suivantes, nous allons, pour chacun de ces groupes, mettre en œuvre des techniques de réécriture de code applicatif, de surcharge de la bibliothèque standard du langage C, de modification dans le noyau et de changement dans la taille des tampons du système de fichiers pour identifier les causes racines des pics mémoire.

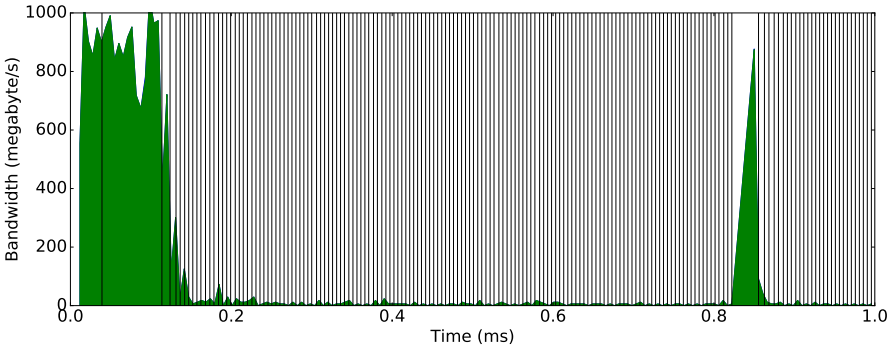


FIGURE 6.3 – Gros plan sur une partie du profil tagué de l'application `Patricia small`

6.2 Impacts des entrée/sortie

Les applications de la suite de test *Mibench* sont, comme décrit précédemment dans la partie 4.3.3, fournies, lorsque c'est nécessaire, avec divers jeux de données sous la forme de fichier texte ou binaire. Ces fichiers sont destinés à émuler les données habituellement fournies sur des plates-formes embarquées par des capteurs matériels qui écrivent directement les informations acquises en mémoire vive. Pour accéder à ces différents fichiers, les applications utilisent les fonctions d'entrée/sortie de la bibliothèque standard du C. Par soucis de réalisme, nous avons donc placé, de la même manière que dans nos travaux réalisés précédemment dans la partie 4.3.3, les données applicatives dans un système de fichiers monté en mémoire principale *Temporary File System*.

Nous allons, dans les sections suivantes, effectuer une étude détaillée des pics de consommation mémoire provoqués par les fonctions d'entrée/sortie, pour ensuite, proposer diverses solutions afin d'y remédier.

6.2.1 Analyse du problème

Le tableau 6.1 liste, pour chacune des applications ayant des pics mémoire, les fonctions d'entrée/sortie à l'origine de ces pics que nous avons identifiées à l'aide de notre mécansime de tags.

La bibliothèque standard du C GNU [128] effectue une distinction entre les fonctions d'accès aux fichiers dites « bas niveau » qui sont utilisées par l'application `adpcm` et les fonctions d'accès dites « haut niveau » qui sont utilisées par les autres applications. Les fonctions d'accès de « haut niveau » opèrent sur des **flux**, les données accédées par le programme en lecture ou en écriture depuis un périphérique sont copiées dans un tampon associé à chaque flux créé qui a pour vocation de servir de cache entre les données applicatives et le système de fichiers. Les fonctions d'accès bas niveau, quant à elles, ne possèdent pas un tel tampon mais disposent de fonctionnalités de configuration plus avancée s'appliquant sur les descripteurs de fichiers.

Applications	Fonctions								
	open	read	close	fopen	fclose	fread	fwrite	fgetc	fgets
ADPCM small encode	x	x	x						
ADPCM small decode	x	x	x						
Patricia small				x	x				x
Rijndael decode				x	x	x	x		
Rijndael encode				x	x	x	x		
Sha small				x	x	x			
Susan small e				x	x	x		x	
Susan small c				x	x	x		x	
Susan large c				x	x	x		x	

Table 6.1 – Fonctions d'entrée/sortie sources de pics de bande passante mémoire

6.2.1.1 Fonctions d'entrée/sortie « haut niveau »

En étudiant le profil mémoire tagué des différentes applications génératrices de pics de consommation mémoire et qui font usage de fonctions d'accès « haut niveau », nous avons constaté que les pics mémoire applicatifs se produisaient quasi systématiquement pendant un appel aux fonctions d'écriture ou de lecture dans les fichiers d'entrée ou de sortie des applications (`fread`, `fwrite`, `fgets`). Mais si les pics correspondent à de tels appels, nous avons aussi observé, à l'inverse, que tous les appels ne généraient pas de pics.

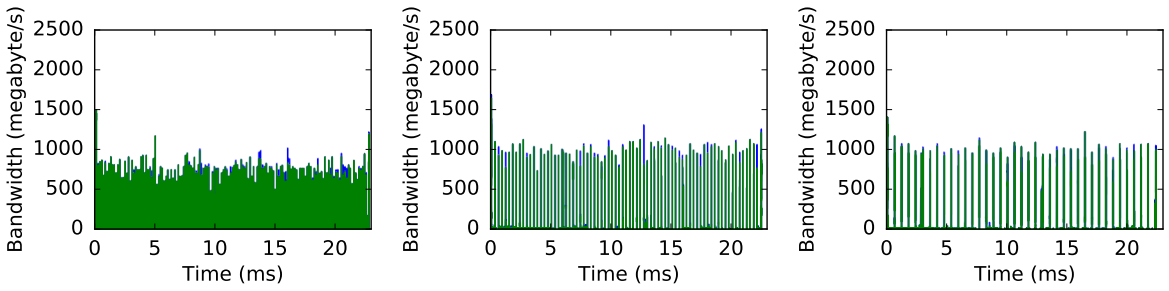
Nous avons, pour étudier plus en profondeur l'impact des fonctions « haut niveau » sur les pics mémoire applicatifs, décidé de réaliser une étude détaillée de l'application `Rijndael small encode`, qui utilise ce type de fonctions pour accéder à des fichiers aussi bien en lecture qu'en écriture. L'algorithme de cette application repose avant tout sur une boucle principale au sein de laquelle le programme acquiert et copie en mémoire, depuis un fichier, des blocs de donnée d'une taille de 16 octets en utilisant la fonction `fread`, effectue des calculs de chiffrement dessus, puis les écrits dans un fichier de sortie en utilisant la fonction `fwrite`.

Afin de faciliter l'analyse du profil mémoire, nous avons donc décidé d'étudier séparément le trafic mémoire généré par la fonction `fread` du trafic mémoire généré par la fonction `fwrite`, en commentant les appels à ladite fonction d'écriture. Une telle modification dans le code ne change pas outre mesure le fonctionnement de l'applicatif, les impacts d'un tel changement se limitant à l'absence d'écriture des données chiffrées dans le fichier de sortie.

Appels à la fonction `fread` La figure 6.4b qui présente le profil mémoire de l'application ainsi modifiée, révèle un faible trafic mémoire de fond régulièrement entrecoupé de pics de bande passante. Nous mesurons, en moyenne, 79 pics mémoire supérieurs à 600 MB/s pour chaque exécution de l'application sur un total de 19489 appels à la fonction `fread` ce qui donne une moyenne d'un pic de consommation mémoire tous les 246.7 appels à la fonction `fread`. La taille du fichier d'entrée étant de 311,824 octets, nous observons donc qu'un pic de consommation se produit en moyenne tous les 3947 octets lus par l'application.

La fonction `fread(void *ptr, size_t size, size_t nmemb, FILE *stream)` lit depuis le flux `stream` `nmemb` éléments chacun d'eux ayant une taille de `size` octets, les copie dans le tampon applicatif `ptr` et avance le descripteur de fichier du flux de `size * nmemb` octets. Lorsque les données requises lors de l'appel à la fonction ne sont pas déjà présentes dans le tampon interne associé au flux, un bloc de donnée de la taille adéquate est lu depuis le fichier associé au flux et copié dans le tampon interne. Une deuxième copie est alors effectuée depuis le tampon interne vers le tampon applicatif. Nous avons, en utilisant la fonction `__fbufsize`, récupéré la taille allouée par défaut pour le tampon du flux d'entrée, à savoir 4096 octets, soit une taille quasi équivalente au volume de donnée lu entre deux pics (3947 octets). Nous supposons donc que les pics de consommation se produisent lorsque la fonction `fread` remplis le tampon interne du flux d'entrée en lisant un nouveau blocs de 4096 octets depuis le fichier.. Dans notre cas, le fichier d'entrée étant présent dans un système de fichier *TMPFS* monté en mémoire, le remplissage du tampon interne du flux provoque une copie depuis la mémoire allouée au *TMPFS* vers le tampon interne ce qui provoque le pic de consommation mémoire. Tous les appels effectués à `fread` font également une copie depuis le tampon interne au flux de lecture vers le tampon applicatif. Cette copie ne génère aucun trafic mémoire, les deux tampons étant chargés dans les caches du processeur.

Afin de tester et de valider notre postulat, nous avons décidé d'augmenter la taille du tampon interne du flux des données d'entrée de l'application en utilisant la fonction `setvbuf` fournie par la libc. Cette modification devrait, d'une part, réduire le nombre de fois où le tampon interne est rempli, se traduisant par une diminution du nombre de pics mémoire dans le profil, et, d'autre part, accroître la hauteur et la durée des pics de bande passante mémoire, le remplissage d'un plus gros tampon générant plus de trafic mémoire. La figure 6.4 montre les profils mémoire de l'application `Rijndael small encode` exécutée avec de multiples tailles de tampons. Nous pouvons observer que l'augmentation de la taille des tampons internes au flux réduit le nombre de pics mémoire, augmente leur durée, et augmente légèrement leur hauteur lorsque la taille de tampon varie de 2 kilooctets à 4 kilooctets validant ainsi notre hypothèse.



(a) 2 KB (b) 4 KB (Défaut) (c) 8 KB

FIGURE 6.4 – Profils mémoire de `Rijndael small encode` exécutée avec les appels à la fonction `fwrite` inhibés et en faisant varier les tailles du tampon de cache pour le flux des données d'entrée de l'application

Appels à la fonction `fwrite` Afin d'étudier le trafic mémoire généré par la fonction d'écritures `fwrite`, nous ne pouvions utiliser la technique précédemment mise en œuvre pour ana-

lyser le trafic mémoire généré par la fonction de lecture des données, à savoir commenter les appels à la fonction `fread`, car ils sont nécessaires au bon fonctionnement du programme. Nous avons donc opté pour une approche différente consistant à surcharger l'appel à la fonction `fread` par un appel à la fonction de lecture `read`, qui ne dispose pas de tampons cache, de telle sorte à lisser le trafic mémoire généré par la fonction de lecture. La fonction effectue alors une lecture depuis le fichier en *TMPFS* par paquet de 16 octets au lieu de 4096 octets. Nous avons ensuite évalué la pertinence de cette approche en traçant, dans la figure 6.5, le profil mémoire de l'application ainsi modifiée tout en laissant l'appel à la fonction `fwrite` commenté. Nous pouvons voir que la quasi-totalité des pics de bande passante mémoire supérieur à 500 Mb/s ont disparus, les pics restants étant dus au système d'exploitation (Voir partie 6.3).

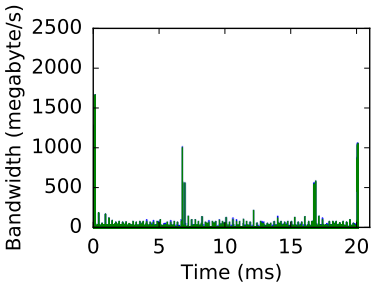
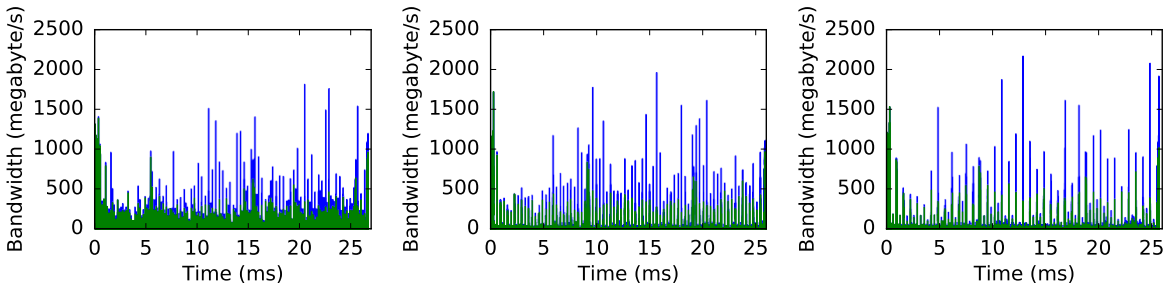


FIGURE 6.5 – Profil mémoire de `Rijndael small encode` exécutée avec les appels à la fonction `fwrite` inhibés et avec une fonction de lecture, des données d'entrée de l'application, « bas niveau »

Le trafic mémoire généré par la lecture des données d'entrée de l'application étant désormais lissé, nous avons dé-commenté les appels à la fonction `fwrite` et tracé le profil résultant de cette opération dans la figure 6.6b. Nous avons calculé, qu'en moyenne pour chaque exécution, 83 pics supérieur à 300 MB/s apparaissent régulièrement. La taille du fichier de sortie étant de 311,856 octets, nous observons qu'un pic de bande passante se produit en moyenne à chaque fois qu'un bloc de 3757 octets a été écrit par l'application. Le volume de données moyen écrit entre deux pics générés par les appels à la fonction `fwrite` étant proche de celui qui transite (3947 octets) entre deux pics générés par les appels à la fonction `fread`, nous supposons donc que l'explication que nous avons émise pour justifier de la présence de pics lors des appels à la fonction `fread` s'applique également pour les appels à la fonction `fwrite`. Lorsque la fonction `fwrite` est appelée et que le tampon est vide les données écrites sont copiées depuis le tampon utilisateur vers le tampon de flux des données de sortie, cette copie ne génère aucun trafic mémoire, les deux tampons étant régulièrement utilisés et sont en conséquence chargés dans les caches du processeur. Lorsque le tampon interne au flux de sortie est plein, les données sont alors vidées vers le système de fichier en mémoire provoquant les pics observés. Nous avons donc, dans la figure 6.6, modifié la taille du tampon du flux de sortie et nous observons une réduction du nombre de pics de bande passante mémoire validant ainsi notre hypothèse.

Nous avons effectué le même type d'expérimentations sur les autres applications qui utilisent des fonctions d'entrée/sortie « haut niveau » et nous avons observé le même comportement, nous permettant ainsi de généraliser les conclusions tirées de notre analyse de l'application `Rijndael small encode` à l'ensemble des autres applications utilisant des fonctions



(a) 2 KB (b) 4 KB (Défaut) (c) 8 KB

FIGURE 6.6 – Profil mémoire de `Rijndael small encode` exécutée avec une fonction de lecture, des données d'entrée de l'application, « bas niveau » en faisant varier les tailles de tampon pour le flux de sortie

d'entrée/sortie du même type.

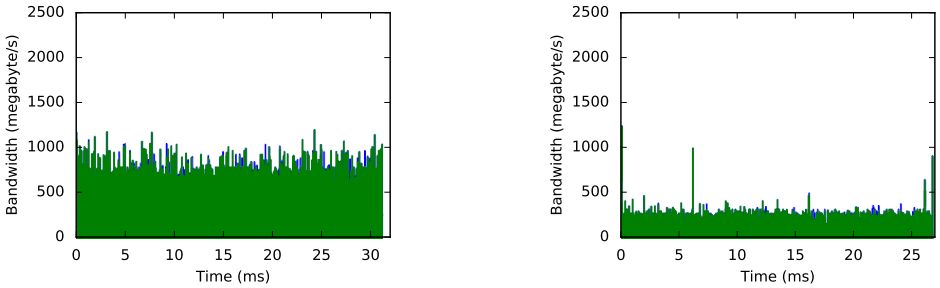
6.2.1.2 Fonctions d'entrée/sortie « bas niveau »

Nous allons maintenant prêter attention à la seule application qui à la fois génère des pics de bande passante et utilise des fonctions d'accès « bas niveau » aux fichiers : `ADPCM`.

`ADPCM` est une application de compression de données avec perte qui, au sein d'une boucle principale, fait un appel à la fonction `read` pour récupérer un bloc de données, effectue des calculs dessus puis les écrits sur la sortie standard en faisant un appel à la fonction `write`.

La taille des blocs bruts pris en entrée par le programme `ADPCM small encode` et émis en sortie par le programme `ADPCM small decode` est de 2 kilooctets tandis que la taille des blocs encodés pris en entrée par le programme `ADPCM small decode` et émis en sortie par le programme `ADPCM small encode` est de 500 octets.

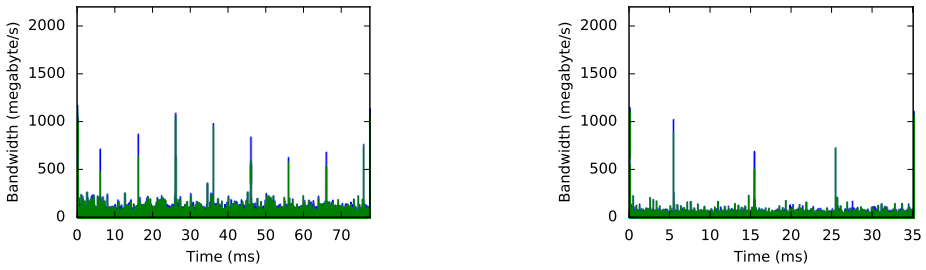
Pour évaluer les impacts des appels effectués à la fonction `read`, nous avons commenté les appels à la fonction `write` et observé le profil mémoire applicatif résultant 6.7 qui affiche un nombre de pics de mémoire identique au nombre d'appels à la fonction `read`, suggèrant que les pics sont dus aux copies depuis le système de fichier *TMPFS* vers les tampons applicatifs.



(a) `ADPCM small encode` (b) `ADPCM small decode`

FIGURE 6.7 – Profils mémoire d'`ADPCM small` avec les écritures commentées et une résolution d'1 µs

Nous avons validé cette hypothèse, en réduisant, dans la figure 6.8, la taille du tampon applicatif de 2 kilooctets à 100 octets. Ce changement accroît substantiellement le temps d'exécution de l'application, mais permet d'éliminer la plupart des pics de consommation mémoire en lecture.



(a) `ADPCM small encode` (b) `ADPCM small decode`

FIGURE 6.8 – Profils mémoire d'`ADPCM small` effectués avec les écritures commentées et un petit tampon de flux de lecture avec une résolution d'1 µs

Nous avons effectué la même opération de réduction de taille des tampons en décommettant les écritures. Nous n'avons pas observé de diminution des pics de mémoire en écriture, l'utilisation d'écritures vers la sortie standard étant la cause des pics de consommation mémoire additionnels.

6.2.2 Conclusion & solutions

Les pics de consommation mémoire des fonctions d'accès « haut niveau » peuvent être générés par l'utilisation de tampons à vocation de cache. Lorsque les applications effectuent des lectures ou des écritures à une granularité plus faible que celle utilisée dans les tampons internes, les données sont lues ou écrites par bloc de même taille que celle utilisée dans les tampons internes ce qui provoque ou amplifie des pics de consommation mémoire.

De notre point de vue, les fonctions d'accès « bas niveau » sont représentatives des accès mémoires qui pourraient être effectués par des applications embarquées qui accèdent à des données générées par des périphériques externes. Les opérations d'entrée/sortie en elle-même pouvant être, au choix, faite par le *Direct Memory Acess* ou par une boucle de copie exécutée par le CPU. D'un autre coté, les fonctions « haut niveau » d'entrée/sortie génèrent des pics additionnels qui ne sont pas souhaitables dans un contexte embarqué. Nous considérons que ces pics proviennent d'un problème de conception de MiBench.

En revanche, les pics de bande passante mémoire générés par les fonctions d'accès « bas niveau » ou par des appels à des fonctions d'accès « haut niveau » en effectuant des accès à des blocs de données de grande taille proviennent de la conception même des applications qui ont besoin d'un important volume de données. Seul une re-conception du logiciel pour lisser les accès sur une période plus longue permet d'éviter l'occurrence de tels pics.

6.3 Impacts du système d'exploitation

Nous avons remarqué que les profils (Figure 6.5 et 6.8), des applications modifiées pour éliminer les pics de consommation mémoire issus des fonctions d'entrée/sortie, contiennent encore des pics de bande passante qui apparaissent régulièrement toutes les 10 ms. L'apparition de tels pics, qui se distinguent par leurs régularités, au sein des multiples applications nous laisse à penser qu'ils sont issus d'une source exogène aux programmes applicatifs. Or, le système d'exploitation étant le seul logiciel capable de prendre la main sur les applications exécutées, nous l'avons donc suspecté d'être à l'origine des pics de consommation mémoire additionnels.

6.3.1 Analyse du problème

Pour étudier plus en profondeur la provenance de ces pics commun à tous les programmes, nous avons profilé l'application présentée dans la partie 6.1.1.2 et dont la faible empreinte mémoire permet de faire ressortir de tels pics. La figure 6.9 affiche le profil mémoire de trois exécutions différentes de cette application. Nous pouvons observer que si les pics mémoire apparaissent régulièrement, leur emplacement varie selon l'exécution de l'application confirmant ainsi notre hypothèse sur l'origine exogène de ces pics. L'apparente régularité des pics nous a amené à porter notre attention sur le mécanisme d'horloge, du système d'exploitation, utilisé pour cadencer les différents évènements.

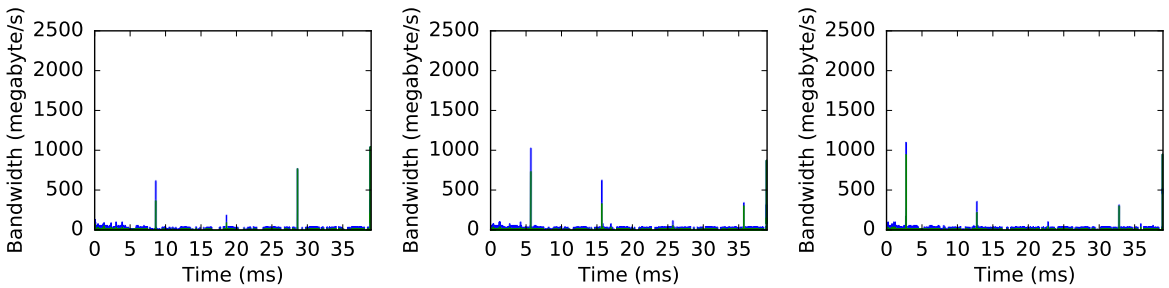


FIGURE 6.9 – Profils mémoire de plusieurs exécutions de l'application à faible empreinte mémoire

La fréquence de l'horloge utilisée par Linux est définie dans le noyau, à la compilation, par l'option de configuration `CONFIG_HZ`, qui est fixée par défaut à 100 Hz, générant ainsi une interruption toutes les 10 ms. Nous avons, pour confirmer nos suppositions, recompilé le noyau Linux pour profiler notre application à faible empreinte mémoire avec une valeur de cadencement de l'horloge fixée à 50 Hz. Le profil mémoire, ainsi obtenu et présenté dans la figure 6.10, montre une augmentation des délais inter-pics qui passe de 10 µs pour une fréquence de 100 Hz à 20 µs pour une fréquence de 50 Hz.

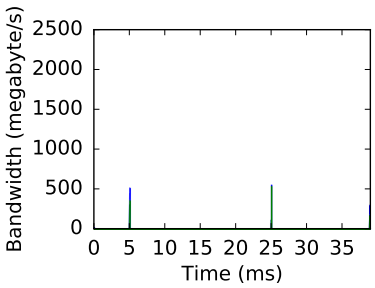


FIGURE 6.10 – Profil mémoire de l'application à faible empreinte mémoire avec un noyau dont l'horloge est cadencée à 50 Hz

6.3.2 Conclusion & solutions

Dans un système d'exploitation Linux traditionnel, chaque CPU est périodiquement interrompu par des interruptions horloges utilisées aussi bien par l'ordonnanceur, pour évaluer quel processus doit être ordonnancé, que pour mettre à jour des mécanismes de synchronisation (*read-copy-update*) ou maintenir le système d'exploitation dans un état cohérent. Les nombreuses opérations effectuées dans le gestionnaire d'interruptions sont la cause des pics de consommation mémoire.

Pour diminuer l'impact de ce mode de fonctionnement, aussi bien en termes de consommation énergétique que de performances pour les systèmes virtualisés, une première option de configuration `CONFIG_NO_HZ` a été rajoutée qui, lorsqu'elle est activée, permet d'arrêter l'horloge lorsque le CPU est *idle*. En revanche, la suppression totale des *ticks* est plus ardue et une première implémentation, soumise à de nombreuses restrictions sur son fonctionnement, a pu être réalisée dans une version de test du noyau Linux.

6.4 Impacts des algorithmes applicatifs

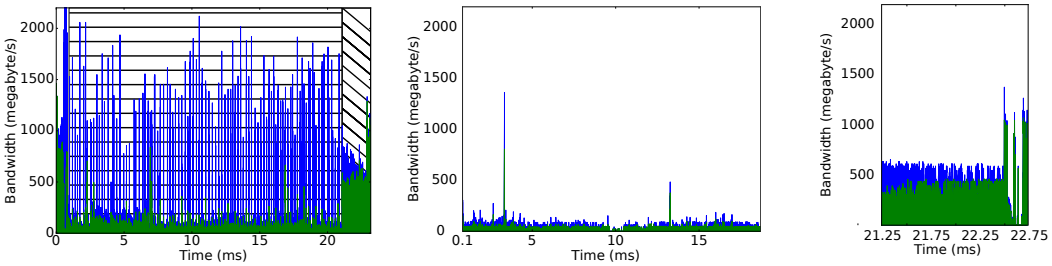
La majorité des applications, qui contiennent des pics de consommation mémoire, exécutent de manière cyclique un même pattern de code produisant ainsi un profil mémoire répétitif. Pour de telles applications, les pics de bande passante sont provoqués par les appels effectués à des fonctions d'entrée/sortie et par les effets de bords du système d'exploitation.

D'autres applications contiennent des pics de bande passante issus directement de leur code source. Nous allons donc, dans une première partie, effectuer une étude détaillée de l'application `Susan large -c` pour étudier les causes logicielles à l'origine de ces pics, puis dans une deuxième partie, proposer diverses solutions pour les éliminer.

6.4.1 Analyse du problème

`Susan large -c` est une application de reconnaissance d'images dont les profils affichés dans la figure 6.2 a-c, affichent des pics de hauteur variable dont la hauteur décroît au fur et à mesure que les intervalles d'échantillonnage diminuent. Pour comprendre la raison de ce

comportement, nous avons utilisé le mécanisme de tags pour identifier les différentes parties de l'application source des pics mettant ainsi en évidence trois différentes phases dans l'exécution de l'application (Figure 6.11a).



(a) Phases

(b) 2e phase de l'application sans écriture

(c) 3e phase de l'application sans écriture

FIGURE 6.11 – Profils mémoire d'étude de l'application `Susan large -c`

6.4.1.1 Première phase

La première phase, d'une durée de 0,1 ms, contient le code de lecture de l'image analysée par l'application qui est, par un appel à la fonction `fread`, copié en mémoire générant ainsi des pics de consommation mémoire.

6.4.1.2 Deuxième phase

Dans la deuxième phase, d'une durée de 20 ms, nous pouvons observer un faible trafic mémoire de fond en lecture d'où émerge régulièrement des pics en écritures de taille et d'amplitude variable. Nous avons utilisé le mécanisme de tags préalablement mis en œuvre pour identifier la partie du code source à l'origine de tels pics. Nous avons ainsi pu isoler une boucle au sein de laquelle l'application itère séquentiellement sur le tableau contenant l'image chargée en mémoire. Le corps de ladite boucle contient une condition complexe qui, lorsqu'elle est évaluée à vrai, sauvegarde des valeurs dans trois différents tableaux d'entiers, chacun d'entre eux ayant la même taille que l'image traitée par l'application. Les opérations d'écritures effectuées dans les tableaux ne sont pas contiguës en mémoires, la distance moyenne entre deux écritures effectuée séquentiellement étant de 287.5 octets, avec un écart type de 565.8 révélant ainsi une importante variation dans la distance entre chaque accès mémoire. A partir du code applicatif, nous avons supputé que le trafic de fond en lecture observé au sein du profil mémoire est généré par les itérations sur le tableau d'octets tandis que les pics de consommation en écritures proviennent des opérations de stockage effectués dans les tableaux d'entiers.

Afin de tester notre hypothèse concernant l'origine des pics de bande passante en écriture nous avons commenté le code applicatif utilisé pour écrire dans les vecteurs d'entiers. Une telle modification n'impacte pas fonctionnellement le comportent de l'application durant la deuxième phase, les valeurs écrites en mémoire étant uniquement utilisées dans la troisième phase de l'application. Nous pouvons observer, dans la figure 6.11b, que le profil mémoire de

la deuxième phase de l'application ainsi modifiée ne contient plus les pics de consommation mémoire générés par les écritures confirmant ainsi notre postulat de départ.

6.4.1.3 Troisième phase

La troisième phase de l'application génère un trafic mémoire en lecture qui oscille aux alentour de 550 MB/s. Le code source de cette phase comporte deux boucles imbriquées utilisées pour parcourir de manière non séquentielle un tableau d'entiers. Le corps de la boucle interne contient une condition qui, lorsqu'elle est valide, sauvegarde des valeurs dans un tableau de structures contiguës en mémoire. Pour différencier l'impact des écritures dans le tableau de structures de l'impact des lectures non séquentielles, nous avons décidé de commenter les opérations d'écritures. Le trafic mémoire se stabilise alors à 550 MB/s (figure 6.11c). En outre, nous avons modifié le code pour supprimer les accès non contigus et avons observé une importante décroissance du trafic mémoire mesuré.

6.4.2 Conclusion & solutions

La plupart des pics de consommation mémoire de l'application `Susan large -c` proviennent de deux sources : les fonctions d'entrée/sortie utilisées pour charger l'image dans les tampons applicatifs et les accès mémoire effectués de manière non contiguës qui provoquent des défauts de caches.

La modification, à la main, du code source applicatif pour éliminer les pics de consommation mémoire, en favorisant l'utilisation de la localité du cache, est extrêmement ardue, le code source de `susan` pouvant contenir jusqu'à 21 niveaux d'imbrications.

Nous pensons donc que la prise en compte des contraintes de consommation mémoire doit être faite dès l'étape de conception de l'application ou bien alors être effectuée ultérieurement par des outils de *refactoring* automatique du code source.

6.5 Conclusion

Nous avons dans ce chapitre, développé une version modifiée de notre profileur mémoire qui utilise les compteurs matériels pour capturer les instructions logicielles qui sont à la source du trafic mémoire global observé. Nous avons ensuite évalué l'impact de notre outil sur les applications étudiées pour éviter que celui-ci n'altère involontairement le comportement des applicatifs faussant ainsi les profils mémoire obtenus. Nous avons estimé que les perturbations générée par le profileur restaient en dessous d'un seuil tolérable pour nos travaux. Ensuite, nous avons tracé le profil mémoire de 13 applications de la suite de test MiBench que nous avons précédemment sélectionnées comme étant temporellement impactée par la consommation mémoire générée par d'autres programmes exécutés en parallèle.

Les profils mémoire résultant révèlent que plus de la moitié des applications choisies au sein de la suite de test *Mibench* contiennent des pics de consommation mémoire. Nous avons mis en place une méthodologie pour identifier l'origine de ces pics en établissant un lien entre le code

source des applicatifs et les profils mémoire correspondants. Nous avons ainsi pu effectuer une classification des pics mémoire en trois catégories : les pics mémoire engendrés par la bibliothèque standard C, les pics mémoire générés par le système d'exploitation et les pics mémoires issus du code source des applicatifs.

Nous avons mis en œuvre des techniques de réécriture de code applicatif, de surcharge de la bibliothèque standard du C, de modification de la taille des tampons des fonctions d'accès aux fichiers et de re-compilation du noyau Linux pour identifier la provenance des pics de consommation mémoire observés au sein des applicatifs.

Nous avons ainsi pu établir que les pics de consommation mémoire générés par les appels à la bibliothèque standard C proviennent des fonctions d'entrée/sortie qu'elles soient « bas niveau » ou qu'elles utilisent des tampons pour cacher en mémoire les données accédées. Les fonctions « haut niveau » d'entrée/sortie génèrent des pics de bande passante mémoire par le remplissage des tampons internes aux flux, tandis que les fonctions d'accès « bas niveau » génèrent des pics mémoires à chaque accès de taille importante effectué aux fichiers. Nous proposons, dans le premier cas, de supprimer le tampon intermédiaire des fonctions bas niveaux pour diminuer voir éliminer les pics de consommation mémoire, l'élimination des pics de mémoire issus des fonctions bas niveaux ne pouvant être effectuée par une re-conception du code applicatif pour lisser les accès sur de plus grande période.

Nous avons également montré que le système d'exploitation et sa gestion du temps était générateur de pics. Sous certaines conditions, imposées par le système, il serait envisageable d'utiliser une version *tickless* de l'OS.

Enfin, nous avons par une étude détaillée de `Susan large -c`, étudié les causes des pics applicatifs. La seule solution que nous préconisons pour éliminer ces pics passe par une réécriture du code applicatif, qu'elle soit manuelle ou automatisée.

Conclusion

Si l'utilisation d'architectures fédérées a suffi pour assurer le développement des systèmes mécatroniques des premiers véhicules, la demande insatiable des consommateurs pour ajouter de nouvelles fonctionnalités les a poussées dans leurs derniers retranchements. L'émergence de calculateurs multi-cœurs toujours plus puissants permet aujourd'hui d'envisager le déploiement d'une nouvelle architecture opérationnelle : une architecture intégrée basée sur la virtualisation. Elle vise à regrouper, sur une même carte plus performante, des logiciels temps réel provenant de systèmes mécatroniques et des logiciels *best effort* multimédia.

Cette solution a de nombreux avantages en termes de réductions des coûts, d'intégration logicielle et de consommation énergétique, mais elle présente l'inconvénient d'abolir la séparation physique entre les logiciels, qui était effective dans les architectures fédérées préexistantes. L'hyperviseur doit donc prendre le relais pour partager le matériel et garantir l'isolation spatiale et temporelle entre les systèmes.

Il est ainsi possible de mettre en place des mécanismes permettant d'assurer le partage de certaines ressources matérielles (CPU, GPU, ...), tout en garantissant une isolation stricte. Cependant, comme nous l'avons vu dans cette thèse, le partage de la mémoire et de son bus d'accès continue encore aujourd'hui de poser des problèmes et reste l'un des défis pour un passage aux architectures virtualisées multi-cœurs. En effet, les mécanismes déjà existants (MMU, MPU) permettent d'assurer un partitionnement spatial mais ne règlent pas les problèmes d'interférences et de contention sur le bus. Tout accès mémoire effectué par une tâche temps réel peut se voir retardé par les accès effectués par les tâches *best effort* provoquant au final le non-respect des échéances temporelles.

Si des solutions de partage temporel strict peuvent être appliquées, elles ne permettent pas une utilisation efficace des processeurs multi-cœurs. Dans ce manuscrit, je me suis attaché à proposer une nouvelle solution permettant de maximiser l'utilisation des multi-cœurs tout en respectant les contraintes des applications temps réel. Ces travaux ont été réalisés dans le cadre d'une collaboration entre le département sûreté et fiabilité de Renault et le laboratoire LIP6. La solution proposée tient donc compte de contraintes industrielles fortes détaillées dans le chapitre 1.

Contributions

Afin de bien comprendre le problème de contention mémoire, j'ai, en préambule de ce manuscrit (Chapitre 2), présenté une analyse des différents composants électroniques du système mémoire, en m'attachant à étudier les phénomènes d'interférences. J'ai aussi confronté (Chapitre 4) cette analyse aux caractéristiques de la carte i.MX6, sujet d'étude de mes travaux.

Afin de quantifier l'ampleur du problème, j'ai ensuite réalisé une plate-forme expérimentale permettant de mesurer le problème de contention mémoire. Cette plate-forme repose sur des sondes noyau ainsi que sur un injecteur de charges paramétrable. Lors du développement de cette plate-forme, je me suis attaché à configurer la carte de telle sorte à minimiser les impacts des interférences inter-cœurs sur les applications. Une campagne de tests de plusieurs semaines a été menée sur les applications de la suite *Mibench* et a montré que les problèmes d'interférences étaient significatifs sur notre carte (jusqu'à 183% de retard). Un mécanisme de régulation est donc nécessaire pour pouvoir mettre en place une architecture intégrée sur cette carte multi-cœurs.

Partant de ce constat, j'ai réalisé un état de l'art (Chapitre 3) des solutions de la littérature permettant de borner, d'éliminer ou de réduire les impacts de la contention sur le système mémoire. Dans cet état de l'art, je me suis attaché à étudier l'adéquation des solutions proposées avec les contraintes industrielles, à savoir : modification des applications interdites, documentation partielle limitant les approches par calculs de WCET et carte à bas coûts, offrant peu de compteurs matériels. Malgré les nombreuses solutions de qualité proposées dans le domaine, une nouvelle approche est alors apparue comme nécessaire.

Lors de cette étude, les approches de régulation se sont avérées comme étant les plus adéquates pour répondre à nos contraintes. J'ai ainsi proposé une nouvelle approche (Chapitre 5) de régulation reposant sur deux mécanismes. Dans une première étape hors ligne, effectuée une fois par application temps réel, je propose une technique de caractérisation du comportement de la plate-forme matérielle en présence de contention mémoire. Ces résultats permettent de générer un oracle capable d'évaluer les ralentissements de l'application temps réel en fonction de la bande passante globale mesurée. La deuxième étape, exécutée en ligne, repose sur un régulateur intégré au noyau. Les applications temps réel sont alors exécutées en parallèle de nos applications *best effort* et notre régulateur mesure, périodiquement, la consommation mémoire. A chaque mesure, ce dernier utilise l'oracle pour détecter les surcoûts temporels qui impactent la tâche temps réel. Les tâches *best effort* sont suspendues lorsque le ralentissement estimé devient incompatible avec les échéances de la tâche temps réel. Elles ne reprendront leurs exécutions que lorsque l'application temps réel aura terminé son traitement.

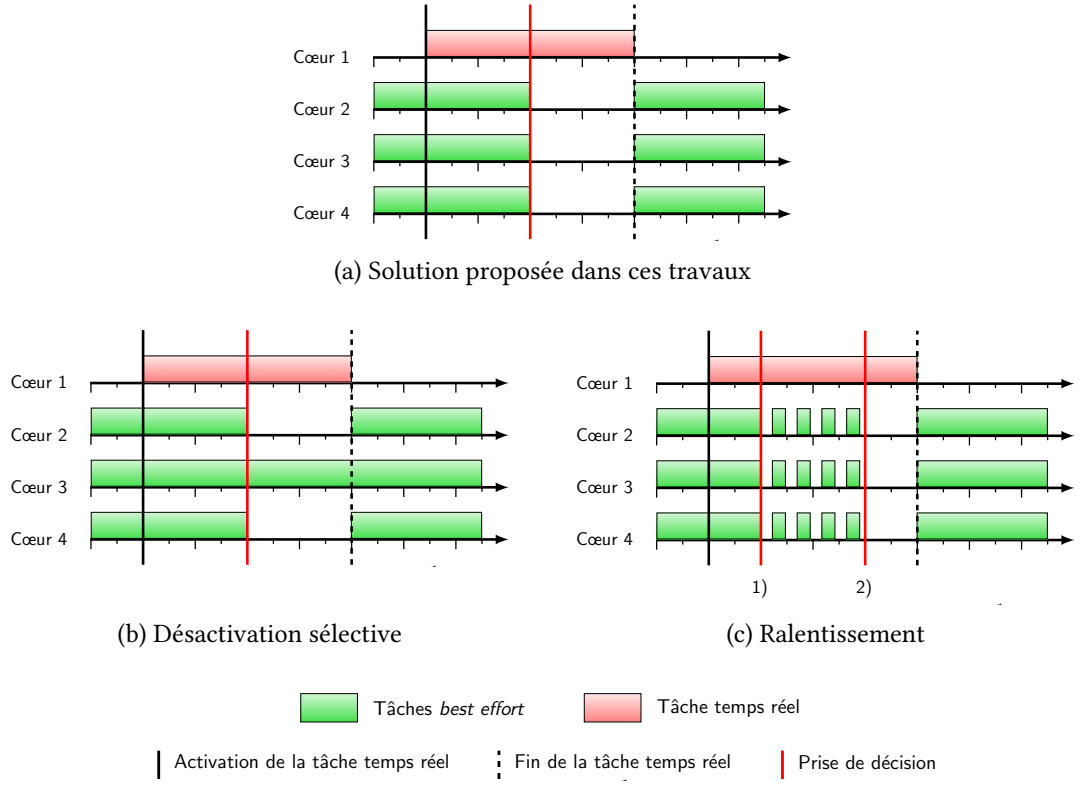
En reprenant la plate-forme expérimentale développée au début de mes travaux, j'ai mené une campagne d'expérimentations en utilisant, comme applications temps réel, des programmes issus de la suite de tests *Mibench*. Au final, la solution développée atteint, pour 7 des 13 applications testées, jusqu'à 70% de parallélisme dès lors que les tâches *best effort* génèrent peu de consommation mémoire. Le parallélisme atteint jusqu'à 100% lorsque les tâches *best effort* n'interfèrent pas avec les applications.

Enfin, dans une dernière contribution (Chapitre 6), j'ai mis à profit notre plate-forme ex-

périmentale pour effectuer une nouvelle étude dans le but de comprendre les raisons de la sensibilité des applications temps réel face à la contention mémoire. J'ai ainsi identifié trois causes principales des pics de consommation mémoire : entrées/sorties, système d'exploitation et algorithmes applicatifs. Pour chacune d'entre elles, je propose des pistes permettant de désensibiliser, à la conception, les applications temps réel.

Perspectives

Notre prototype de recherche a été conçu en utilisant un système d'exploitation en lieu et place d'un hyperviseur qui était alors indisponible. Nous prévoyons donc, à moyen terme, d'intégrer notre mécanisme au sein de l'hyperviseur qui sera choisi par Renault.



(a) Solution proposée dans ces travaux

(b) Désactivation sélective

(c) Ralentissement

FIGURE 6.12 – Solutions futures proposées

Les travaux présentés dans cette thèse nécessitant une analyse manuelle du code applicatif, il pourrait être pertinent de chercher à automatiser la détection de phases. Ces travaux pourraient être effectués en combinant une analyse du code source, afin de détecter les motifs de code répétitifs, avec du profilage mémoire pour identifier les phases. Il pourrait également être intéressant d'établir une corrélation entre les données d'entrée des applications et les tables de surcoûts. En effet les variations des données d'entrée peuvent impacter la bande passante mémoire générée et au final les ralentissements mesurés. Notre approche considérant le pire cas, elle se doit de prendre en compte les pires bandes passantes mesurées. Il serait donc intéressant

de pouvoir effectuer une telle distinction.

Si ces améliorations portent plus sur l'oracle et le mécanisme de profilage, il pourrait être intéressant, à plus long terme, de retravailler le mécanisme de contrôle et, plus particulièrement, la méthode à utiliser pour arrêter les tâches *best effort*. Ainsi, nous pourrions améliorer aussi bien le choix des cœurs à suspendre que le mécanisme utilisé pour les stopper.

Dans notre solution actuelle, lorsque des ralentissements sont estimés, nous désactivons comme illustré dans la figure 6.12a, l'ensemble des cœurs *best effort* qu'ils soient ou non, fortement consommateurs de mémoire. Cette approche suppose une charge homogène de la part des cœurs *best effort*, ce qui dans la pratique peut s'avérer inexacte. Une approche plus sélective, illustrée dans la figure 6.12b, pourrait être mise en œuvre afin de ne suspendre que les cœurs réellement à l'origine de la contention mémoire. Il serait possible d'utiliser une heuristique, basée sur les compteurs des défauts des caches L1 qui donnent une estimation grossière du trafic mémoire généré par chacun des cœurs, pour inférer les cœurs les plus consommateurs.

Notre approche repose sur un arrêt complet des tâches *best effort* en cas de contention mémoire. Cela peut conduire à une remise en cause de leur interactivité lorsque notre mécanisme se déclenche trop souvent. Une nouvelle approche pourrait être utilisée afin de réduire préventivement la pression exercée par ces applications sur le bus mémoire. L'idée est de limiter les interférences avec l'application temps réel, dès les premiers signes de contention, pour éviter d'avoir à les stopper totalement (figure 6.12c). En pratique, l'idéal serait de baisser la fréquence de l'horloge des cœurs qui ordonnancent ces applications lorsque de la contention mémoire est détectée. Malheureusement, sur notre carte, comme sur d'autres cartes COTS, il est impossible de baisser la fréquence à ce niveau de granularité. Une approche alternative serait de ralentir les applications *best effort* en utilisant des interruptions horloges, ces tâches s'exécutant alors par intermitence. Parallèlement à ce mécanisme de ralentissement, il sera tout de même nécessaire de conserver le mécanisme d'arrêt total, ce dernier s'activant lorsque l'approche par ralentissement se montre insuffisante.

Publications

Antoine BLIN, Cédric COURTAUD, Julien SOPENA, Julia LAWALL, Gilles MULLER ; *Understanding the Memory Consumption of the MiBench Embedded Benchmark* ; Conference on Networked Systems (NETYS'16), May 2016, Marakech, Morocco. 2016.

Antoine BLIN, Cédric COURTAUD, Julien SOPENA, Julia LAWALL, Gilles MULLER ; *Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System* ; 28th EUROMICRO Conference on Real-Time Systems (ECRTS'16), Jul 2016, Toulouse, France.

Antoine BLIN, Julien SOPENA, Gilles MULLER, Youssef LAAROUCHI ; *Contrôle de la bande passante mémoire dans les systèmes à criticité mixte par sous-réservation* ; ComPAS 2014 - Conférence d'informatique en Parallélisme, Architecture et Systeme, Neuchâtel, Switzerland (2014)

Antoine BLIN, Youssef LAAROUCHI, Philippe QUERE ; *Fault-Injection using Virtualization for Critical Software Validation in Automotive* ; ERTS 2014 Embedded Real Time Software and Systems

Bibliographie

[1] Robin J Adair. **A virtual machine system for the 360/40**. International Business Machines Corporation, Cambridge Scientific Center, 1966.

[2] Benny Akesson, Kees Goossens et Markus Ringhofer. « Predator : a predictable SDRAM memory controller ». In : **Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis**. ACM. 2007, p. 251–256.

[3] Benny Akesson, Liesbeth Steffens, Eelke Strooisma et Kees Goossens. « Real-time scheduling using credit-controlled static-priority arbitration ». In : **2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications**. IEEE. 2008, p. 3–14.

[4] Ahmed Alhammad et Rodolfo Pellizzoni. « Schedulability analysis of global memory-predictable scheduling ». In : **Proceedings of the 14th International Conference on Embedded Software**. ACM. 2014, p. 20.

[5] **Appareils de traitement de l'information - Caractéristiques des perturbations radioélectriques - Limites et méthodes de mesure**. norme. 'Association française de normalisation, 2012.

[6] ARM. **ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition**. C.b. Nov. 2012.

[7] ARM. **ARM Generic Interrupt Controller Architecture Specification 2.0**. B.b. Juil. 2013.

[8] ARM. **Cortex A9 MPCore, Technical Reference Manual**. Revision : r4p1. Juin 2012.

[9] ARM. **Cortex-A Series, Programmer s Guide**. Version : 3.0. Juin 2012.

[10] ARM. **Cortex-A9, Technical Reference Manual**. Revision : r4p1. Juin 2012.

[11] ARM. **SABRE Lite Hardware User Manual**. 1.1. Juil. 2011.

[12] Abu Asaduzzaman, Fadi N Sibai et Manira Rani. « Improving cache locking performance of modern embedded systems via the addition of a miss table at the L2 cache level ». In : **Journal of Systems Architecture** 56.4 (2010), p. 151–162.

[13] **AUTOSAR**. url : `http://www.autosar.org/`.

[14] Stanley Bak, Gang Yao, Rodolfo Pellizzoni et Marco Caccamo. « Memory-aware scheduling of multicore task sets for real-time systems ». In : **Embedded and Real-Time**

Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on. IEEE. 2012, p. 300–309.

[15] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan et Peter Marwedel. « Scratchpad memory : design alternative for cache on-chip memory in embedded systems ». In : **Proceedings of the tenth international symposium on Hardware/software codesign**. ACM. 2002, p. 73–78.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt et Andrew Warfield. « Xen and the art of virtualization ». In : **ACM SIGOPS Operating Systems Review** 37.5 (2003), p. 164–177.

[17] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch et Bruno Zoppis. « The VMware mobile virtualization platform : is that a hypervisor in your pocket ? » In : **ACM SIGOPS Operating Systems Review** 44.4 (2010), p. 124–135.

[18] **BMW Targets 2020 for Self-Driving Cars**. url : http://www.autoguide.com/auto-news/2013/02/bmw-targets-2020-for-self-driving-cars.html.

[19] Frédéric Boniol, Hugues Cassé, Eric Noulard et Claire Pagetti. « Deterministic execution model on cots hardware ». In : **Architecture of Computing Systems–ARCS 2012**. Springer, 2012, p. 98–110.

[20] Karel A Brookhuis, Dick De Waard et Wiel H Janssen. « Behavioural impacts of advanced driver assistance systems–an overview ». In : **European Journal of Transport and Infrastructure Research** 1.3 (2001), p. 245–253.

[21] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum et Monica S Lam. « Compiler-directed page coloring for multiprocessors ». In : **ACM SIGPLAN Notices**. T. 31. 9. ACM. 1996, p. 244–255.

[22] Edouard Bugnion, Scott Devine, Kinshuk Govil et Mendel Rosenblum. « Disco : Running Commodity Operating Systems on Scalable Multiprocessors ». In : **ACM Trans. Comput. Syst.** 15.4 (nov. 1997), p. 412–447.

[23] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman et Edward Y Wang. « Bringing virtualization to the x86 architecture with the original vmware workstation ». In : **ACM Transactions on Computer Systems (TOCS)** 30.4 (2012), p. 12.

[24] **Car 2 Car consortium**. url : https://www.car-2-car.org/.

[25] **Car 2 Car consortium**. url : http://www.mobileye.com/.

[26] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel et Heiko Falk. « A unified WCET analysis framework for multicore platforms ». In : **ACM Transactions on Embedded Computing Systems (TECS)** 13.4s (2014), p. 124.

[27] Sudipta Chattopadhyay, Abhik Roychoudhury et Tulika Mitra. « Modeling shared cache and bus in multi-cores for timing analysis ». In : **Proceedings of the 13th international workshop on software & compilers for embedded systems**. ACM. 2010, p. 6.

[28] Yu Chen, Wenlong Li, Changkyu Kim et Zhizhong Tang. « Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy ».

In : **Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on**. IEEE. 2009, p. 1–11.

[29] Ali Chousein et Rabi N Mahapatra. « Fully associative cache partitioning with don't care bits for real-time applications ». In : **ACM SIGBED Review** 2.2 (2005), p. 35–38.

[30] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoit Triquet et Reinhard Wilhelm. « Predictability considerations in the design of multi-core embedded systems ». In : ().

[31] **Daimler aims to launch self-driving car by 2020**. url : http://www.reuters.com/article/us-autoshow-frankfurt-daimler-selfdrive-idUSBRE98709A20130908.

[32] **Device Virtualization**. HARMAN. url : http://www.redbend.com/en/products/device-virtualization.

[33] Marco Di Natale. « Design and development of component-based embedded systems for automotive applications ». In : **Reliable Software Technologies–Ada-Europe 2008**. Springer, 2008, p. 15–29.

[34] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian et Haibing Guan. « High performance network virtualization with SR-IOV ». In : **Journal of Parallel and Distributed Computing** 72.11 (2012), p. 1471–1480.

[35] Guy Durrieu, Madeleine Faugere, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti et Wolfgang Puffitsch. « Predictable flight management system implementation on a multicore processor ». In : **Embedded Real Time Software (ERTS'14)**. 2014.

[36] Stephen A Edwards et Edward A Lee. « The case for the precision timed (PRET) machine ». In : **Proceedings of the 44th annual Design Automation Conference**. ACM. 2007, p. 264–265.

[37] Christian El Salloum, Martin Elshuber, Oliver Höftberger, Haris Isakovic et Armin Wasicek. « The ACROSS MPSoC–A new generation of multi-core processors designed for safety–critical embedded systems ». In : **Microprocessors and Microsystems** 37.8 (2013), p. 1020–1032.

[38] **Elon Musk : Don't fall asleep at the wheel for another 5 years**. url : http://www.cnet.com/roadshow/news/elon-musk-sees-autonomous-cars-ready-sooner-than-previously-thought/.

[39] **Embedded Linux Conference 2013 - KEYNOTE Google's Self Driving Cars**. url : https://www.youtube.com/watch?v=7Yd9Ij0INX0.

[40] **Emissions in the automotive sector**. url : http://ec.europa.eu/growth/sectors/automotive/environment-protection/emissions/index_en.htm.

[41] Stuart Fisher. **Certifying Applications in a Multi-Core Environment : The World's First Multi-Core Certification to SIL 4**. Rapp. tech. Sysgo, 2013.

[42] **Ford Is Building An Autonomous, Self-Driving Car For the Masses**. url : http://www.ibtimes.com/ford-building-autonomous-self-driving-car-masses-2250032.

[43] Marc Gatti, Xavier Jean, Laurent Pautet, Thomas Robert et David Faura. « Ensuring robust partitioning in multicore platforms for IMA systems ». In : **2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)**. IEEE. 2012, p. 1–18.

[44] **General Motors President sees self-driving cars by 2020**. URL : http://www.cnet.com/roadshow/news/general-motors-president-sees-self-driving-cars-by-2020/.

[45] Olaf GIETELINK, Jeroen PLOEG, Bart DE SCHUTTER et Michel VERHAEGEN. « Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations ». In : **Vehicle System Dynamics** 44.7 (2006), p. 569–590.

[46] Sylvain GIRBAL, Xavier JEAN, Jimmy LE RHUN, Daniel GRACIA PÉREZ et Marc GATTI. « Deterministic platform software for hard real-time systems using multi-core COTS ». In : **2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)**. IEEE. 2015, p. 8D4–1.

[47] Sylvain GIRBAL, Xavier JEAN, Jimmy LE RHUN, Daniel GRACIA PÉREZ et Marc GATTI. « Deterministic platform software for hard real-time systems using multi-core COTS ». In : **2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)**. IEEE. 2015, p. 8D4–1.

[48] Thomas GLEIXNER et Douglas NIEHAUS. « Hrtimers and beyond : Transforming the linux time subsystems ». In : **Proceedings of the Linux symposium**. T. 1. Citeseer. 2006, p. 333–346.

[49] Robert P GOLDBERG. **Architectural principles for virtual computer systems**. Rapp. tech. DTIC Document, 1973.

[50] Sven GOOSSENS, Benny AKESSON, Martijn KOEDAM, Ashkan Beyranvand NEJAD, Andrew NELSON et Kees GOOSSENS. « The CompSOC design flow for virtual execution platforms ». In : **Proceedings of the 10th FPGAworld conference**. ACM. 2013, p. 7.

[51] **GPS**. URL : http://www.gps.gov/systems/gps/modernization/sa/.

[52] Giovani GRACIOLI, Ahmed ALHAMMAD, Renato MANCUSO, Antônio Augusto FRÖHLICH et Rodolfo PELLIZZONI. « A survey on cache management mechanisms for real-time embedded systems ». In : **ACM Computing Surveys (CSUR)** 48.2 (2015), p. 32.

[53] Nan GUAN, Martin STIGGE, Wang YI et Ge YU. « Cache-aware scheduling and analysis for multicores ». In : **Proceedings of the seventh ACM international conference on Embedded software**. ACM. 2009, p. 245–254.

[54] Andreas GUSTAVSSON, Andreas ERMEDAHL, Björn LISPER et Paul PETTERSSON. « Towards WCET analysis of multicore architectures using UPPAAL ». In : **OASIcs-OpenAccess Series in Informatics**. T. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[55] Matthew R GUTHAUS, Jeffrey S RINGENBERG, Dan ERNST, Todd M AUSTIN, Trevor MUDGE et Richard B BROWN. « MiBench : A free, commercially representative embedded benchmark suite ». In : **Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on**. IEEE. 2001, p. 3–14.

[56] Andreas HANSSON, Kees GOOSSENS, Marco BEKOOIJ et Jos HUISKEN. « CoMPSoC : A template for composable and predictable multi-processor system on chips ». In : **ACM Transactions on Design Automation of Electronic Systems (TODAES)** 14.1 (2009), p. 2.

[57] Damien HARDY, Thomas PIQUET et Isabelle PUAUT. « Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches ». In : **Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE**. IEEE. 2009, p. 68–77.

[58] Gernot Heiser. « Hypervisors for consumer electronics ». In : **Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE**. IEEE. 2009, p. 1–5.

[59] Gernot Heiser. « The role of virtualization in embedded systems ». In : **Proceedings of the 1st workshop on Isolation and integration in embedded systems**. ACM. 2008, p. 11–16.

[60] Gernot Heiser et Ben Leslie. « The OKL4 microvisor : convergence point of microkernels and hypervisors ». In : **Proceedings of the first ACM asia-pacific workshop on Workshop on systems**. ACM. 2010, p. 19–24.

[61] Dandan Huan, Zusong Li, Weiwu Hu et Zhiyong Liu. « Processor directed dynamic page policy ». In : **Advances in Computer Systems Architecture**. Springer, 2006, p. 109–122.

[62] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park et Chul-Ryun Kim. « Xen on ARM : System virtualization using Xen hypervisor for ARM-based secure mobile phones ». In : **Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE**. IEEE. 2008, p. 257–261.

[63] **INTEGRITY Multivisor**. Green Hills. url : `http://www.ghs.com/products/rtos/integrity_virtualization.html`.

[64] Kenji Ishikawa, Michima Ogawa, Shigetoshi Azuma et Tooru Ito. « Map navigation software of the electro-multivision of the'91 Toyoto Soarer ». In : **Vehicle Navigation and Information Systems Conference, 1991**. T. 2. IEEE. 1991, p. 463–473.

[65] **ISO 26262 Véhicules routiers – Sécurité fonctionnelle**. norme. International Organization for Standardization, 2008.

[66] **ISO 7736 :1984 Véhicules routiers – Autoradio avec montage par l'avant**. norme. International Organization for Standardization, 1984.

[67] Ravi Iyer. « CQoS : a framework for enabling QoS in shared caches of CMP platforms ». In : **Proceedings of the 18th annual international conference on Supercomputing**. ACM. 2004, p. 257–266.

[68] **Joint Electron Device Engineering Council**. JEDEC. url : `https://www.jedec.org/`.

[69] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay et Abhik Roychoudhury. « Bus-aware multicore WCET analysis through TDMA offset bounds ». In : **2011 23rd Euromicro Conference on Real-Time Systems**. IEEE. 2011, p. 3–12.

[70] Kurt Keutzer, Jan M Rabaey, A Sangiovanni-Vincentelli et al. « System-level design : orthogonalization of concerns and platform-based design ». In : **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on** 19.12 (2000), p. 1523–1543.

[71] **Kia joins other automakers, says self-driving cars are on the way**. url : `http://www.csmonitor.com/Business/In-Gear/2016/0107/Kia-joins-other-automakers-says-self-driving-cars-are-on-the-way`.

[72] Hyoseung Kim, Arvind Kandhalu et Ragunathan Rajkumar. « A coordinated approach for practical OS-level cache management in multi-core real-time systems ». In : **2013 25th Euromicro Conference on Real-Time Systems**. IEEE. 2013, p. 80–89.

[73] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu et Ragunathan Rajkumar. « Bounding memory interference delay in COTS-based multi-core systems ». In : **2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)**. IEEE. 2014, p. 145–154.

[74] Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M Petters et Henrik Theiling. « Multicore in real-time systems–temporal isolation challenges due to shared resources ». In : **Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems**. 2014.

[75] Yogen Krishnapillai, Zheng Pei Wu et Rodolfo Pellizzoni. « A rank-switching, open-row DRAM controller for time-predictable systems ». In : **2014 26th Euromicro Conference on Real-Time Systems**. IEEE. 2014, p. 27–38.

[76] Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, Matthieu Roy et Fabian Vargas. « Monitoring on-line timing information to support mixed-critical workloads ». In : **IEEE Real-Time Systems Symposium 2013**. 2013, p. 9–10.

[77] Angeliki Kritikakou, Claire Pagetti, Olivier Baldellon, Matthieu Roy et Christine Rochange. « Run-time control to increase task parallelism in mixed-critical systems ». In : **Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on**. IEEE. 2014, p. 119–128.

[78] Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal et Daniel Gracia Pérez. « Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems ». In : **Proceedings of the 22nd International Conference on Real-Time Networks and Systems**. ACM. 2014, p. 139.

[79] Benjamin Lesage, Damien Hardy et Isabelle Puaut. « Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. » In : **18th International Conference on Real-Time and Network Systems**. 2010, p. 2283.

[80] Benjamin Lesage, Isabelle Puaut et André Seznec. « PRETI : Partitioned Real-Time shared cache for mixed-criticality real-time systems ». In : **Proceedings of the 20th International Conference on Real-Time and Network Systems**. ACM. 2012, p. 171–180.

[81] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra et Abhik Roychoudhury. « Timing analysis of concurrent programs running on shared cache multi-cores ». In : **Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE**. IEEE. 2009, p. 57–67.

[82] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang et P Sadayappan. « Gaining insights into multicore cache partitioning : Bridging the gap between simulation and real systems ». In : **2008 IEEE 14th International Symposium on High Performance Computer Architecture**. IEEE. 2008, p. 367–378.

[83] John S. Liptay. « Structural aspects of the System/360 Model 85, II : The cache ». In : **IBM Systems Journal** 7.1 (1968), p. 15–21.

[84] Chun Liu, Anand Sivasubramaniam et Mahmut Kandemir. « Organizing the last line of defense before hitting the memory wall for CMPs ». In : **Software, IEE Proceedings-**. IEEE. 2004, p. 176–185.

[85] Isaac Liu, Jan Reineke et Edward A Lee. « A PRET architecture supporting concurrent programs with composable timing properties ». In : **2010 Conference Record of the**

Forty Fourth Asilomar Conference on Signals, Systems and Computers. IEEE. 2010, p. 2111–2115.

[86] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen et Chengyong Wu. « A software memory partition approach for eliminating bank-level interference in multicore systems ». In : **Proceedings of the 21st international conference on Parallel architectures and compilation techniques**. ACM. 2012, p. 367–376.

[87] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo et Rodolfo Pellizzoni. « Real-time cache management framework for multi-core architectures ». In : **Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th**. IEEE. 2013, p. 45–54.

[88] David Marshall. **White paper : Understanding Full Virtualization, Paravirtualization, and Hardware Assist**. Rapp. tech. VMware, 2007.

[89] **Mécatronique – Vocabulaire**. norme NF E 01-010. Association française de normalisation, nov. 2008.

[90] Stefan Metzlaff, Sascha Uhrig, Jörg Mische et Theo Ungerer. « Predictable dynamic instruction scratchpad for simultaneous multithreaded processors ». In : **Proceedings of the 9th workshop on MEmory performance : DEaling with Applications, systems and architecture**. ACM. 2008, p. 38–45.

[91] Charlie Miller et Chris Valasek. « A survey of remote automotive attack surfaces ». In : ().

[92] **MPPA : The Supercomputing on a chip solution**. Kalray. url : `http://www.kalrayinc.com/kalray/products/%5C#processors`.

[93] Sai Prashanth Muralidhara, Mahmut Kandemir et Padma Raghavan. « Intra-application cache partitioning ». In : **Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on**. IEEE. 2010, p. 1–12.

[94] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir et Thomas Moscibroda. « Reducing memory interference in multicore systems via application-aware memory channel partitioning ». In : **Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture**. ACM. 2011, p. 374–385.

[95] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers et Rich Uhlig. « Intel Virtualization Technology : Hardware Support for Efficient Processor Virtualization. » In : **Intel Technology Journal** 10.3 (2006).

[96] **NHTSA Announces Final Rule Requiring Rear Visibility Technology**. url : `http://www.nhtsa.gov/About+NHTSA/Press+Releases/2014/NHTSA+Announces+Final+Rule+Requiring+Rear+Visibility+Technology`.

[97] Roman Obermaisser, Christian El Salloum, Bernhard Huber et Hermann Kopetz. « From a federated to an integrated automotive architecture ». In : **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems** 28.7 (2009), p. 956.

[98] **OKL4 Microvisor**. General Dynamics Broadband. url : `http://www.ok-labs.com/products/okl4-microvisor`.

[99] Marco Paolieri, Eduardo Quinones, Francisco J Cazorla et Mateo Valero. « An analyzable memory controller for hard real-time CMPs ». In : **IEEE Embedded Systems Letters** 1.4 (2009), p. 86–90.

[100] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo et Russell Kegley. « A predictable execution model for COTS-based embedded systems ». In : **Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE**. IEEE. 2011, p. 269–279.

[101] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo et Lothar Thiele. « Worst case delay analysis for memory interference in multicore systems ». In : **Proceedings of the Conference on Design, Automation and Test in Europe**. European Design et Automation Association. 2010, p. 741–746.

[102] Jan Pelzl, Marko Wolf et Thomas Wollinger. **Virtualization technologies for cars**. Rapp. tech. Technical Report 2008, escrypt GmbH-Embedded Security, 2008.

[103] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham et Rebecca Isaacs. « 30 seconds is not enough ! : a study of operating system timer usage ». In : **EuroSys**. 2008.

[104] **PikeOS**. SYSGO AG. url : `http://www.sysgo.com`.

[105] Gerald J Popek et Robert P Goldberg. « Formal requirements for virtualizable third generation architectures ». In : **Communications of the ACM** 17.7 (1974), p. 412–421.

[106] Isabelle Puaut et Christophe Pais. « Scratchpad memories vs locked caches in hard real-time systems : a quantitative comparison ». In : **2007 Design, Automation & Test in Europe Conference & Exhibition**. IEEE. 2007, p. 1–6.

[107] Moinuddin K Qureshi et Yale N Patt. « Utility-based cache partitioning : A low-overhead, high-performance, runtime mechanism to partition shared caches ». In : **Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture**. IEEE Computer Society. 2006, p. 423–432.

[108] Nauman Rafique, Won-Taek Lim et Mithuna Thottethodi. « Architectural support for operating system-driven CMP cache management ». In : **Proceedings of the 15th international conference on Parallel architectures and compilation techniques**. ACM. 2006, p. 2–12.

[109] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim et Edward A Lee. « PRET DRAM controller : Bank privatization for predictability and temporal isolation ». In : **Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis**. 2011, p. 99–108.

[110] Jakob Rosen, Alexandru Andrei, Petru Eles et Zebo Peng. « Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip ». In : **Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International**. IEEE. 2007, p. 49–60.

[111] Rusty Russell. « virtio : towards a de-facto standard for virtual I/O devices ». In : **ACM SIGOPS Operating Systems Review** 42.5 (2008), p. 95–103.

[112] Kristian Sandstrom, Aneta Vulgarakis, Markus Lindgren et Thomas Nolte. « Virtualization technologies in embedded real-time systems ». In : **Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on**. IEEE. 2013, p. 1–8.

[113] Alberto Sangiovanni-Vincentelli et Marco Di Natale. « Embedded system design for automotive applications ». In : **Computer** 10 (2007), p. 42–51.

[114] Abhik Sarkar, Frank Mueller et Harini Ramaprasad. « Predictable task migration for locked caches in multi-core systems ». In : **ACM SIGPLAN Notices** 46.5 (2011), p. 131–140.

[115] Martin Schoeberl. « Jop : A java optimized processor for embedded real-time systems ». Thèse de doct. Technical University of DenmarkDanmarks Tekniske Universitet, Department of Informatics et Mathematical ModelingInstitut for Informatik og Matematisk Modellering.

[116] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann et al. « T-CREST : Time-predictable multi-core architecture for embedded systems ». In : **Journal of Systems Architecture** 61.9 (2015), p. 449–471.

[117] Freescale Semiconductor. **CoreLink Level 2 Cache Controller L2C-310, Technical Reference Manual**. Revision : r3p3. 2012.

[118] Freescale Semiconductor. **i.MX 6Dual/6Quad Applications Processor Reference Manual**. Revision : 1. Avr. 2013.

[119] Freescale Semiconductor. **i.MX 6Dual/6Quad Automotive and Infotainment Applications Processors**. Revision : 2. Avr. 2013.

[120] Freescale Semiconductor. **Leopard MPC5643L Microcontroller**.

[121] Kang G Shin et Parameswaran Ramanathan. « Real-time computing : A new discipline of computer science and engineering ». In : ().

[122] Shekhar Srikantaiah, Mahmut Kandemir et Mary Jane Irwin. « Adaptive set pinning : managing shared caches in chip multiprocessors ». In : **ACM SIGARCH Computer Architecture News**. T. 36. 1. ACM. 2008, p. 135–144.

[123] Udo Steinberg et Bernhard Kauer. « NOVA : a microhypervisor-based secure virtualization architecture ». In : **Proceedings of the 5th European conference on Computer systems**. ACM. 2010, p. 209–222.

[124] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian et Al Davis. « Micro-pages : increasing DRAM efficiency with locality-aware data placement ». In : **ACM Sigplan Notices** 45.3 (2010), p. 219–230.

[125] Vivy Suhendra et Tulika Mitra. « Exploring locking & partitioning for predictable shared caches on multi-cores ». In : **Proceedings of the 45th annual Design Automation Conference**. ACM. 2008, p. 300–303.

[126] Karthik T Sundararajan, Timothy M Jones et Nigel P Topham. « RECAP : region-aware cache partitioning ». In : **2013 IEEE 31st International Conference on Computer Design (ICCD)**. IEEE. 2013, p. 294–301.

[127] David Tam, Reza Azimi, Livio Soares et Michael Stumm. « Managing shared L2 caches on multicore systems in software ». In : **Workshop on the Interaction between Operating Systems and Computer Architecture**. Citeseer. 2007, p. 26–33.

[128] **The GNU C Library**. GNU. url : `https://www.gnu.org/software/libc/manual/html_mono/libc.html`.

[129] **The Motor Industry Software Reliability Association**. url : `http://www.misra.org.uk/`.

[130] Kun Tian, Yaozu Dong et David Cowperthwaite. « A full GPU virtualization solution with mediated pass-through ». In : **2014 USENIX Annual Technical Conference (USENIX ATC 14)**. 2014, p. 121–132.

[131] **Toyota on the fast track to autonomous vehicles**. URL : http://news.yahoo.com/toyota-fast-track-autonomous-vehicles-114318802.html;_ylt=AwrC0F9Q8BNW0DsAFMLQtDMD;_ylu=X3oDMTByOHZyb21tBGNvbG8DYmYxBHBvcwMxBHZ0aWQDBHNlYwNzcg--.

[132] **True virtual acceleration with GPUs**. NVIDIA. URL : http://www.nvidia.com/object/grid-technology.

[133] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando Martins, Andrew V Anderson, Steven M Bennett, Alain Kägi, Felix H Leung et Larry Smith. « Intel virtualization technology ». In : **Computer** 38.5 (2005), p. 48–56.

[134] Theo Ungerer, Francisco J Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf et al. « Merasa : Multicore execution of hard real-time applications supporting analyzability ». In : **IEEE Micro** 5.30 (2010), p. 66–75.

[135] Keshavan Varadarajan, SK Nandy, Vishal Sharda, Amrutur Bharadwaj, Ravi Iyer, Srihari Makineni et Donald Newell. « Molecular Caches : A caching structure for dynamic creation of application-specific Heterogeneous cache regions ». In : **Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture**. IEEE Computer Society. 2006, p. 433–442.

[136] Prashant Varanasi et Gernot Heiser. « Hardware-supported virtualization on ARM ». In : **Proceedings of the Second Asia-Pacific Workshop on Systems**. ACM. 2011, p. 11.

[137] **Véhicules routiers – Gestionnaire de réseau de communication (CAN)**. norme. International Organization for Standardization, 2003.

[138] **Véhicules routiers – Local Interconnect Network**. norme. International Organization for Standardization, 2015.

[139] **Véhicules routiers – Système de communications FlexRay**. norme. International Organization for Standardization, 2013.

[140] **VIRTUALIZATION**. Wind River. URL : http://www.windriver.com/products/operating-systems/virtualization/.

[141] Bryan C Ward, Jonathan L Herman, Christopher J Kenna et James H Anderson. « Outstanding paper award Making shared caches more predictable on multicore platforms ». In : **2013 25th Euromicro Conference on Real-Time Systems**. IEEE. 2013, p. 157–167.

[142] Jon Watson. « Virtualbox : bits and bytes masquerading as machines ». In : **Linux Journal** 2008.166 (2008), p. 1.

[143] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister et Christian Ferdinand. « Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems ». In : **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems** 28.7 (2009), p. 966.

[144] Reinhard Wilhelm et Jan Reineke. « Embedded systems : Many cores-Many problems. » In : ().

[145] Maurice V Wilkes. « Slave memories and dynamic storage allocation ». In : **IEEE Transactions on Electronic Computers** 2.EC-14 (1965), p. 270–271.

[146] Zheng Pei Wu, Yogen Krish et Rodolfo Pellizzoni. « Worst case analysis of DRAM latency in multi-requestor systems ». In : **Real-Time Systems Symposium (RTSS), 2013 IEEE 34th**. IEEE. 2013, p. 372–383.

[147] Jun Yan et Wei Zhang. « WCET analysis for multi-core processors with shared L2 instruction caches ». In : **Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE**. IEEE. 2008, p. 80–89.

[148] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti et Marco Caccamo. « Memory-centric scheduling for multicore hard real-time systems ». In : **Real-Time Systems** 48.6 (2012), p. 681–715.

[149] Heechul Yun, Renato Mancuso, Zheng-Pei Wu et Rodolfo Pellizzoni. « PALLOC : DRAM bank-aware memory allocator for performance isolation on multicore platforms ». In : **2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)**. IEEE. 2014, p. 155–166.

[150] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo et Lui Sha. « MemGuard : Memory bandwidth reservation system for efficient performance isolation in multi-core platforms ». In : **Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th**. IEEE. 2013, p. 55–64.

[151] Xiao Zhang, Sandhya Dwarkadas et Kai Shen. « Towards practical page coloring-based multicore cache management ». In : **Proceedings of the 4th ACM European conference on Computer systems**. ACM. 2009, p. 89–102.

[152] Trevor Zink, Frank Maker, Roland Geyer, Rajeevan Amirtharajah et Venkatesh Akella. « Comparative life cycle assessment of smartphone reuse : repurposing vs. refurbishment ». In : **The International Journal of Life Cycle Assessment** 19.5 (2014), p. 1099–1109.