Caractérisation de la sensibilité aux interférences mémoire dans les systèmes temps-réels embarqués sur des plateformes multi-cœurs

Cédric Courtaud

THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité **Informatique**
École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par
Cédric COURTAUD

Pour obtenir le grade de

DOCTEUR de Sorbonne Université

Caractérisation de la sensibilité aux interférences mémoire dans les systèmes temps-réels embarqués sur des plateformes multi-cœurs

---

Soutenue le 28 janvier 2020 devant le jury composé de

Gilles GRIMAUD	*Rapporteur*	Professeur, Université de Lille 1
Daniel HAGIMONT	*Rapporteur*	Professeur, INPT/ENSHEITT
Liliana CUCU-GROSJEAN	*Examinateur*	Chargée de recherche, Inria
Lionel LACASSAGNE	*Examinateur*	Professeur, Sorbonne Université
Isabelle PUAUT	*Examinateur*	Professeur, Université de Rennes 1
Gilles MULLER	*Directeur de thèse*	Directeur de recherche, Inria
Daniel GRACIA PÉREZ	*Encadrant*	Ingénieur de recherche, Thales
Julien SOPENA	*Encadrant*	Maitre de conférence, Sorbonne Université

Remerciements

Beaucoup de personnes ont contribuées directement ou indirectement à rendre ce manuscrit possible. Je tiens ici à les remercier.

Tout d'abords, un grand merci à mes rapporteurs, Gilles Grimaud et Daniel Hagimont, pour le temps qu'ils ont consacré à l'examen de ce document et pour leur commentaires et leur avis. J'addresse également mes remerciements à Liliana Cucu-Grosjean, Lionel Lacassagne et Isabelle Puaut d'avoir accepté de faire partie de mon jury.

Ce document n'aurait jamais vu le jour sans mes encadrants industriels, Xavier Jean puis Daniel Gracia Pérez, et académiques, Gilles Muller et Julien Sopena. Merci à vous pour vos précieux conseils, votre enthousiasme et le soutien que vous m'avez offert au cours de ces années.

Cette thèse CIFRE a été financée par Thales Research & Technology. J'ai passé trois belles années chez Thales, et c'est en grande partie grace aux différentes personnes que j'ai pu y cottoyer. Je remercierais d'abords Marc Gatti pour rendu cette thèse possible. Je remercie ensuite pour leur acceuil les membres de l'équipe LSEC : Phillipe Bonnot, Madeleine Faugère, Sylvain Girbal, Jimmy Le Rhun, Arnaud Grasset, et Mustapha Iguerdane. Merci aussi à mes collègues doctorants de STI pour leur soutien : Baptiste Goupille-Lescar et Christophe Prevot. Enfin je remercie Sébastien Jacq et Kevin Eyssartier pour tout les bons moments que l'on a pu vivre en partageant le même bureau.

Je conserverais aussi un excellent souvenir de mes années au LIP6. Tout d'abords merci à Redha Gouicem, Lucas Serrano, Darius Mercadier, Pierre Nigron, Rémi Oudin, Damien Carver, Yoann Guigoff et Antoine Blin pour avoir contribué à en faire un lieu où il faisait bon venir. Je remercie également les autres membres de l'équipes Whisper, Pierre Evariste Dagand et Julia Lawall, pour les échanges fructueux que nous avons pu avoir au cours des dernières années. Merci aussi à Swann Dubois et Pierre

Sens pour les conseils qu'ils m'ont apportés durant la préparation de la soutenance. Je remercie également les doctorants des équipes Whisper et Delys ainsi que les gens que j'ai pu cotoyer au cours de mes années à l'université. Merci donc à : Ilyas, Gauthier, Arnaud, Francis, Gabriel, Marjorie, Denis, Mathieu, Saalik, Guillaume, Kenza, Louis, Joel, Jonathan, Hakan. Merci également à Dany Richard pour l'aide qu'elle m'as apporté dans les démarches nécesssaires à ma soutenance malgré ma conception assez personnelles des délais réglementaires. Mes excuses aux nombreuses personnes que j'ai surement oubliées.

Mes derniers remerciements iront à ma famille et mes amis proches. Je remercie mes parents, ainsi que mon frère, Fabrice, pour tout leur encouragements et leur soutien. Merci à Loulou et à Juliette de s'être inquiété de ma charge de travail pendant la rédaction de ce document. Je remercierais enfin Stéphane pour avoir si bien su me rappeler qu'il n'y a pas que la thèse dans la vie.

Résumé

Les interférences du système mémoire peuvent entraîner d'importants ralentissements aux applications s'exécutant en parallèle sur les processeurs multi-cœurs COTS. Elles ont pour origine les accès concurrents aux ressources matérielles partagées du système mémoire. L'ampleur des retards causés par ce phénomène s'avère difficile à prédire, faisant des interférences un obstacle majeur à l'adoption des processeurs multi-cœur COTS dans les systèmes temps-réels. Cette thèse est consacrée à la caractérisation de la sensibilité d'une application aux interférences mémoires à partir d'une caractérisation de son comportement exécutée seule. Le but étant de pouvoir déterminer à priori si une application est sensible à ce problème ou non. À l'aide d'un ensemble de microbenchmarks que nous avons préalablement introduit, nous montrons qu'une caractérisation purement quantitative du comportement d'accès à la mémoire caractérise la sensibilité aux interférences de façon très imprécise. Afin de permettre une caractérisation plus précise de la sensibilité, nous introduisons différentes métriques permettant de quantifier des aspects quantitatifs de l'utilisation de la mémoire. Afin de mesurer ces métriques, nous implémentons un prototype de profileur reposant sur des approches d'instrumentation binaire dynamique. En plus de permettre la mesure des aspects qualitatifs, cet outil produit des profils haute résolution permettant de distinguer clairement les différentes phases dans les comportements applicatifs. Enfin, nous utilisons les données issues de nos microbenchmarks pour entraîner un algorithme d'apprentissage automatique selon plusieurs caractérisations. Les résultats expérimentaux montrent des réductions significatives de réduction d'erreur pour la prédiction du retard subi par des applications des suites MIBENCH et PARSEC.

Table des matières

CHAPITRE 1

Introduction

Depuis le milieu des années 2000, les processeurs sont majoritairement conçus selon des architectures multi-cœurs. Cette bascule s'est effectuée pour des raisons techniques, principalement liées à la consommation énergétique [25]. Depuis, la plupart des processeurs équipant les ordinateurs de bureau, les serveurs, mais aussi des terminaux mobiles comme les téléphones intelligents ou les tablettes sont des processeurs multi-cœurs. Cette omniprésence se traduit par une offre pléthorique de processeurs sur le marché de masse (on parlera de composants sur étagère ou COTS [1]). Dans cette offre, on trouve notamment des processeurs alliant embarquabilité (taille réduite et faible consommation énergétique) et performances pour un coût modique. Ces bénéfices rendent ce type de matériel particulièrement attrayant pour répondre aux contraintes de coûts et de performances des prochaines générations de systèmes embarqués temps-réels.

Ces progrès techniques accompagnent nombre d'innovations industrielles, comme celui de la voiture autonome. L'utilisation de multicœurs y permet non seulement de multiplier les fonctionnalités, mais aussi de réduire le nombre de calculateurs. Ce dernier point est important, car il simplifie l'intégration et réduit les coûts. Notons aussi que ce passage aux multicœurs est parfois imposé aux industriels, car les calculateurs mono-cœurs se font de plus en plus rares.

Cette transition n'a pas été sans poser de problèmes, notamment pour les systèmes embarqués temps-réels. En effet, elle se heurte aux exigences de sûreté inhérentes à ces systèmes, et notamment celles concernant la sûreté temporelle. Les applications

1. Component Off The Shelf

temps-réel devant répondre en temps borné, elles ont des échéances à respecter sous peine d'entraîner une défaillance du système. Lors de la conception d'un tel système, il convient donc de garantir que les applications le composant puissent respecter leurs échéances. Pour s'en assurer, l'usage est d'étudier l'ordonnancement des différentes applications du système en considérant leur pire temps d'exécution (WCET [2]). Le calcul du WCET est un problème difficile, supposant une bonne connaissance du matériel sur lequel s'exécute l'application et l'absence d'influence extérieure sur son exécution. Ainsi, un critère souvent associé au choix du matériel pour les systèmes embarqués est le *déterminisme*. On doit pouvoir anticiper le comportement du matériel *dans les pires cas*. Or, les processeurs multi-cœurs COTS, étant destinés au marché de masse, sont conçus pour offrir de bonnes performances en moyenne au détriment du déterminisme.

Parmi les sources d'indéterminisme affectant les processeurs COTS, les travaux présentés dans ce document se concentrent sur celles liées aux partages de ressources matérielles entre les différents cœurs. En effet, le partage de ressources tel que les caches les bus, ou encore les contrôleurs mémoires est une source d'interférences entre les applications s'exécutant en parallèle. Ainsi, sur une machine avec un cache partagé, des données utiles à une application A peuvent être évincées au profit de données d'une application B. Si l'application A accède de nouveau à ces données, cela entraînera un défaut de cache qui n'aurait pas eu lieu si elle s'exécutait seule. De plus, le chargement de données faisant suite à ce défaut peut entraîner une situation similaire pour l'application B. Les accès concurrents aux composants matériels partagés entraînent donc des ralentissements qui s' ils ne sont pas pris en compte peuvent entraîner des dépassements d'échéances. Les interférences posent un problème fondamental pour le calcul du WCET, car celui-ci dépend du comportement des autres applications. L'intégration de ces interférences dans ce calcul pose des problèmes mettant en échec les approches utilisées jusqu'à présent. Ces difficultés sont aussi bien causées par la complexité et l'opacité du matériel ciblé, que par la nature concurrente du problème. Ainsi, pour déterminer le retard qu'une application peut subir à cause des interférences, il faut actuellement étudier son comportement en situation de stress sur les ressources matérielles, ce qui implique d'importantes campagnes de tests.

L'ampleur du retard que peut subir une application à cause des interférences dépend de trois facteurs : les applications s'exécutant en parallèle, le matériel et l'application elle-même. En pratique, il est difficile, voire impossible, de raisonner sur le matériel et les autres applications. Partant de ce constat, la question qui se pose est : peut-on estimer

2. Worst Case Execution Time

la sensibilité d'une application aux interférences uniquement à partir de caractéristiques qui lui sont propres ? Par là, nous entendons les caractéristiques décrivant le comportement de l'application sur le matériel ciblé lorsque celle-ci s'exécute seule. Il s'agit alors de ne pas considérer les aspects concurrents du problème, et, ainsi, d'éviter le recours à des analyses complexes ou à de grands nombres d'expériences.

Ce document apporte des éléments de réponses en cherchant à identifier différents aspects caractérisant le comportement des applications en isolation afin de caractériser leur sensibilité aux interférences. Le but étant de pouvoir caractériser ce comportement suffisamment précisément pour estimer rapidement le retard que peut subir une application. Une telle estimation peut être utile pour dimensionner rapidement un système, ou encore évaluer la pertinence d'une cible en particulier. Les difficultés d'analyses inhérentes aux processeurs COTS nous poussent à adopter une approche boite noire, et donc de limiter autant que possible les hypothèses sur le matériel. Afin de caractériser la sensibilité aux interférences à partir du comportement en isolation, nous conduisons un grand nombre d'expériences préliminaires sur le matériel ciblé dans le but de capturer son comportement.

1.1 Contributions

Les contributions apportées par nos travaux s'articulent autour de trois axes :

Microbenchmarks pour l'analyse d'interférences L'ampleur de l'impact des interférences est directement liée à l'utilisation que font les applications du matériel. Pour analyser le comportement d'une plateforme matérielle, il faut donc pouvoir couvrir un large spectre de cas d'utilisation. Dans ce but, nous avons développé un ensemble de microbenchmarks dont on peut faire varier le comportement selon différents paramètres. Les aspects selon lesquels nous faisons varier ce comportement sont déterminés à partir d'une représentation événementielle de l'utilisation de la mémoire faite par les programmes. Nous montrons que ces microbenchmarks permettent de couvrir un grand nombre de cas différents de sensibilité aux interférences.

L'étude du comportement d'accès à la mémoire Nous souhaitons caractériser le comportement d'accès à la mémoire en isolation des applications afin de caractériser leur sensibilité aux interférences. Dans ce document, nous proposons différentes métriques quantifiant différents aspects de ce comportement. Nous faisons notamment une distinction entre les *métriques quantitatives* quantifiant les aspects

liés à l'intensité de l'utilisation que fait l'application du matériel et les *métriques qualitatives* quantifiant les aspects liés à leur nature. Certaines des métriques que nous avons définies n'étant pas mesurables par des moyens traditionnels, nous avons implanté un prototype de profileur reposant sur la simulation du trafic émis par les applications. Cet outil permet également la génération de *profils haute résolution*, représentant finement l'activité d'un programme, et plus particulièrement les différentes phases de son exécution.

L'inférence de surcoût temporel engendré par les interférences Nous utilisons des approches d'apprentissage automatiques pour construire des fonctions de prédiction du surcoût temporel subi par une application en fonction de son comportement d'accès à la mémoire. L'entraînement de ces prédicteurs est effectué à l'aide de données issues de nos microbenchmarks. Nous évaluons à cette occasion la pertinence des différentes métriques que nous avons préalablement définie. Pour cela, nous mesurons la qualité des prédictions obtenues en employant différents ensembles de métriques. Nous montrons ainsi que l'emploi de métriques qualitatives permet d'atteindre des gains de précisions significatifs (jusqu'à 61,2%).

1.2 Organisation du document

Le corps de ce document est organisé en deux parties et cinq chapitres. La première partie, constituée de deux chapitres, expose en détail le problème que posent les interférences. La seconde partie est constituée des trois chapitres restants, où sont exposées les contributions apportées par nos travaux.

Le contenu des chapitres est organisé de la façon suivante :

Chapitre 2 Nous présentons le problème des interférences sous l'angle du matériel. Après avoir formulé des hypothèses de bases sur les systèmes étudiés, nous présentons les différents canaux d'interférences que l'on peut trouver sur le matériel disponible actuellement. Nous nous pencherons ensuite sur le fonctionnement et les interférences du système mémoire, au cœur de nos préoccupations.

Chapitre 3 Nous exposons les contraintes liées à la conception des systèmes temps-réels et l'impact des interférences sur ces systèmes. Nous présentons, ensuite, un état de l'art des approches permettant de prendre en compte le problème des interférences dans les systèmes temps-réels.

Chapitre 4 Nous présentons une représentation événementielle du trafic mémoire nous permettant de distinguer divers aspects de l'utilisation que fait un programme

du système mémoire. Nous proposons et implantons ensuite un ensemble de microbenchmarks permettant de générer divers scénarios d'utilisation de la mémoire variant selon les aspects illustrés précédemment. Enfin, nous utilisons ces microbenchmarks pour évaluer l'ampleur du phénomène des interférences sur une plateforme multi-cœur COTS utilisée dans l'industrie.

Chapitre 5 Nous nous penchons sur la caractérisation du comportement applicatif. Ainsi, nous définissons diverses métriques en rapport avec la sensibilité supposée aux interférences. Nous faisons la distinction entre des métriques quantitatives caractérisant l'intensité de l'utilisation de la mémoire, et des métriques qualitatives caractérisant leur nature. Enfin, nous présentons un outil de profilage permettant de mesurer ces métriques, ainsi que de découper les applications en phases homogènes.

Chapitre 6 Nous évaluons la pertinence des caractéristiques définies dans le chapitre précédent pour la caractérisation de la sensibilité aux interférences. Dans ce but, nous utilisons un algorithme d'apprentissage automatique pour inférer le retard subi par des applications en caractérisant leur comportement de différentes manières. L'apprentissage est effectué à partir de données issues de nos microbenchmarks. Pour terminer, nous présentons les résultats d'une expérience conduite sur les benchmarks des suites MIBENCH et PARSEC. Ces résultats valident le choix des métriques, le profilage, ainsi que la technique d'apprentissage en permettant d'améliorer très significativement la prédiction de la sensibilité en comparaison d'une approche basée uniquement sur la consommation de la bande passante.

Première partie

Présentation du problème

CHAPITRE 2

Processeurs multi-cœur et interférences

L'utilisation de processeurs multi-cœur COTS dans les nouvelles générations de calculateurs est une piste prometteuse pour accompagner la montée en complexité des systèmes embarqués. Cela pose néanmoins le défi de concilier les exigences de déterminisme propre à des systèmes critiques et temps-réel, et les choix de conception de matériel destiné au marché de masse. D'une part, les systèmes embarqués temps-réels doivent se conformer à un principe d'isolation temporelle. C'est-à-dire que dans un tel système la performance d'une application ne doit pas être affectée par celle des autres. D'autre part, le matériel que nous souhaitons utiliser étant conçu pour offrir de bonnes performances à bas coût, des composants matériels sont partagés entre les cœurs, altérant ainsi la performance des applications s'exécutant en parallèle.

Dans ce chapitre, nous allons étudier en détail la cause de ces *interférences*. Nous commencerons par présenter l'architecture des systèmes embarqués et les différentes hypothèses que nous ferons sur ceux-ci dans la suite de nos travaux. Nous présenterons ensuite les différents canaux d'interférence que l'on trouve sur le matériel COTS. Enfin, nous détaillerons le fonctionnement et les différentes sources d'interférences du système mémoire.

2.1 Architecture des systèmes embarqués

Les systèmes embarqués modernes sont complexes, à l'image de ceux que l'on retrouve dans les avions, les trains ou encore les voitures. Ils impliquent généralement un grand nombre de fonctions différentes, communiquant entre elles par le biais d'un réseau. Des approches d'ingénierie ont été proposées pour faire face à la complexité du développement de logiciel embarqué. Afin de découpler les spécifications fonctionnelles des implantations logicielles et matérielles, un découpage en quatre niveaux d'architectures est préconisé :

- *L'architecture fonctionnelle* a pour objectif de décrire et formaliser les différents services rendus par le système. Ces services sont découpés en *blocs fonctionnels* connectés en réseau. Chaque bloc fonctionnel spécifie une fonctionnalité attendue en faisant abstraction des détails de fonctionnement interne.
- *L'architecture logicielle* spécifie le découpage des différents blocs fonctionnels en composants logiciels.
- *L'architecture matérielle* décrit l'ensemble des composants matériels dans le système. Elle comprend notamment des calculateurs reliés entre eux par des réseaux (tel que CAN [71], LIN [30], FlexRay [65], ou encore AFDX [42]).
- *L'architecture opérationnelle* décrit l'association entre l'architecture logicielle et l'architecture matérielle.

Dans cette section, nous allons positionner nos travaux par rapport à ces quatre niveaux d'architectures, en fonction des tendances actuelles dans l'industrie.

2.1.1 Architecture fonctionnelle

Nous ferons l'hypothèse de systèmes à criticité mixte [97], c'est-à-dire comprenant des fonctions critiques et non critiques. On supposera que les fonctions critiques sont temps-réels, et donc que l'altération de leur comportement temporel peut conduire à une défaillance du système. Les fonctions non critiques ne sont soumises à aucune exigence particulière, et peuvent être assurées en *best effort*. Néanmoins, l'absence de contraintes sur les fonctions non critiques implique également que l'on ne peut pas faire d'hypothèse sur leur comportement.

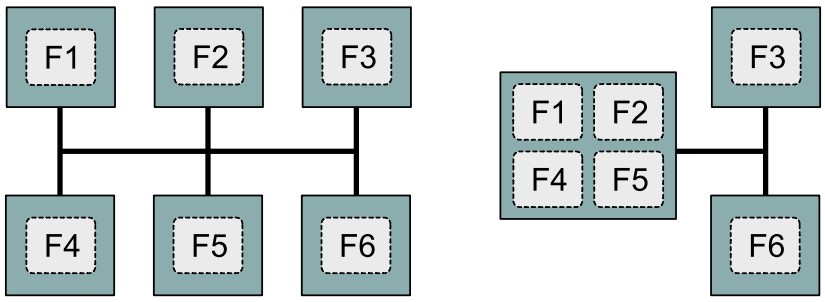
Dans cette thèse, nous nous concentrons uniquement sur l'effet des interférences. Par conséquent, nous supposerons que les fonctions sont indépendantes. Cela se traduit par

l'absence de couplage explicite entre les applications : elles ne communiquent et ne se synchronisent pas.

2.1.2 Architecture opérationnelle

L'architecture opérationnelle décrit comment les différentes briques logicielles sont réparties sur les calculateurs. Il y a deux approches pour effectuer ce placement :

- Les *architectures fédérées* où chaque fonction dispose d'un calculateur dédié. Cela assure un confinement optimal en cas défaillance de l'une d'entre elles. Ce type d'architecture est néanmoins très coûteux à mettre en œuvre, le nombre de calculateurs requis étant proportionnel aux nombres de fonctions du système.
- Les *architectures intégrées* où plusieurs fonctions peuvent partager le même calculateur. Cette approche permet de réduire les coûts en mutualisant les calculateurs. Cela pose néanmoins le problème du confinement des défaillances (fonctionnelles, mais aussi temporelles), à plus forte raison lorsque des fonctions critiques et non critiques partagent le même calculateur.



(a) Système fédéré (b) Système intégré

FIGURE 2.1 – Architectures de systèmes embarqués

La montée en complexité des systèmes embarqués entraine une forte tendance à l'adoption des architectures intégrées. C'est par exemple le cas dans l'avionique avec l'architecture *IMA* [1] [79], utilisée, entre autres, dans l'A380 et l'A350. Dans un système intégré, les fonctions sont autant d'*applications* disposant de leur propre espace d'adressage et communiquant avec le reste du système au moyen d'une interface commune (comme ARINC 653 [80] dans les systèmes avioniques ou encore AUTOSAR [1] dans l'automobile). Un logiciel d'infrastructure (système d'exploitation ou hyperviseur) fait l'interface entre les applications et le matériel.

1. Integrated Modular Avionics

Le logiciel d'infrastructure est notamment en charge de maintenir l'*isolation spatiale et temporelle* des applications. L'isolation spatiale assure que deux applications ne puissent accéder à la mémoire de l'autre. Tandis que l'isolation temporelle assure que la performance d'une application ne puisse être affectée par les autres applications du système.

2.1.3 Architecture matérielle

Nos travaux portent sur l'utilisation de cartes multi-cœurs dans les calculateurs. Ce type de carte permet d'intégrer plus de fonctions au sein d'un même calculateur, en exécutant des fonctions en parallèle. Parmi toute l'offre disponible, nous ciblerons des processeurs en particulier. Il s'agit de petites cartes UMA [2] Achetée sur étagère et destinées au marché de masse. Elles comportent un nombre restreint (entre deux et huit) de cœurs, mais chacun d'eux offre de bonnes performances. La performance des processeurs que nous ciblons repose sur des fonctionnalités matérielles complexes (préchargement de donnée, prédiction de branchement, exécution dans le désordre, etc.), dont le fonctionnement est le plus souvent peu documenté. Par conséquent, nous nous efforcerons de faire le moins d'hypothèses possible sur le matériel. Des exemples processeurs de ce type cité dans la littérature sont le Cortex-A9, ou encore le PowerPC P5040.

Une hypothèse importante pour la suite de nos travaux est la présence de matériel partagé entre les cœurs. Ce partage de matériel est à l'origine d'*interférences*, sur lesquelles nous nous pencherons dans la section suivante. Les interférences sont source de ralentissements particulièrement problématiques dans le cadre de systèmes intégrés, car l'isolation temporelle peut ne pas être respectée. Notons que si certaines architectures partagent des ressources de calcul au niveau des cœurs (technologie HyperThread), ce ne sera pas le cas de celles que nous considérons dans nos travaux. On fera ainsi l'hypothèse qu'un cœur ne peut exécuter au plus qu'un seul fil d'exécution.

2.1.4 Architecture logicielle

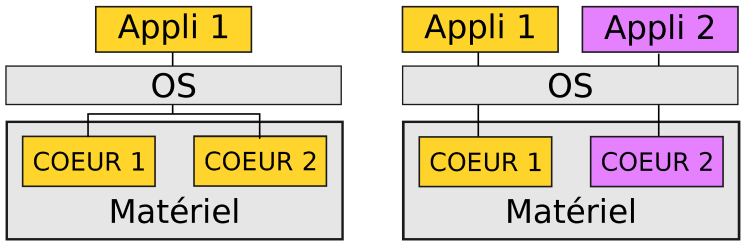
L'exploitation d'une plateforme multi-cœur peut se faire à l'aide de deux modèles de programmation, impactant l'architecture logicielle du système.

- *Modèle symétrique ou Symetric Multi-Processing (SMP)* Une fonction peut s'exécuter sur plusieurs cœurs à la fois. Une fonction est donc découpée en plusieurs compo-

2. Unified Memory Access

sants logiciels qui correspondent à autant de fils d'exécution pouvant avoir besoin d'être synchronisée. Ce modèle permet à une application d'utiliser au mieux le matériel disponible, mais sa mise en œuvre peut nécessiter des adaptations des *applications patrimoniales* si celles-ci ont été développées pour des systèmes mono-cœur.

- *Modèle asymétrique ou Asymetric Multi-Processing (AMP)* Une fonction ne peut s'exécuter que sur un seul cœur à la fois. Ce modèle exploite le matériel en exécutant des fonctions distinctes en parallèle sur les différents cœurs. Par hypothèse, il n'y a pas de synchronisation entre les applications. Il peut néanmoins y en avoir avec le logiciel d'infrastructure. Ce modèle, bien que ne tirant pas parti du parallélisme au niveau d'une application, à l'avantage de ne pas demander la modification du logiciel patrimonial.



(a) Modèle symétrique (SMP) (b) Modèle asymétrique (AMP)

FIGURE 2.2 – Modèles de programmation pour les systèmes multicœurs

Dans nos travaux, nous ferons l'hypothèse de fonctions implantées selon le modèle AMP. Ce choix est motivé par deux raisons. Tout d'abord, les applications patrimoniales étant prépondérantes dans les systèmes embarqués, il s'agit d'un choix réaliste sur le plan industriel. Deuxièmement, comme nos travaux portent sur l'effet des interférences dues au partage de matériel, nous souhaitons exclure autant que possible les retards résultant de l'attente causée par des synchronisations.

2.1.5 Impact sur nos travaux

Dans cette section, nous avons décrit l'architecture générale que l'on retrouve dans les systèmes embarqués complexes. Nous avons aussi formulé un certain nombre d'hypothèses, posant un cadre de travail pour le reste de cette thèse. Ces hypothèses sont les suivantes :

- Les systèmes étudiés sont à *criticité mixte*. Ils comprennent des applications critiques et non-critiques. Les applications critiques sont soumises à des contraintes temps-réels, et on ne peut pas faire d'hypothèses sur les applications non critiques.
- Les systèmes étudiés sont conçus selon une architecture intégrée, un calculateur est partagé par plusieurs fonctions. Un logiciel d'infrastructure est en charge d'assurer l'isolation spatiale et temporelle des différentes fonctions.
- Les calculateurs sont basés sur des processeurs multi-cœur COTS destinés au marché de masse. On se concentre sur des processeurs UMA disposant d'un nombre restreint de cœurs offrant chacun de bonnes performances. Les cœurs partagent du matériel causant ainsi des interférences temporelles. En conséquence, l'isolation temporelle n'est pas assurée lorsque deux applications s'exécutent en parallèle.
- Les fonctions du système sont supposées indépendantes les unes des autres, elles ne communiquent et ne se synchronisent donc pas. Elles sont de plus implantées selon un modèle asymétrique, et ne comportent donc pas de fil d'exécution s'exécutant en parallèle. Par conséquent, nous supposons l'absence de couplage explicite entre les cœurs, et nous concentrons donc uniquement sur l'effet des interférences.

Dans la section suivante, nous allons présenter les différents composants matériels à la source du phénomène d'interférences.

2.2 Canaux d'interférences dans les systèmes multicœurs COTS

Le matériel destiné aux applications grand public est conçu pour offrir de *bonnes performances moyennes*, tout en respectant de fortes contraintes de *coûts*. Afin de réaliser ces objectifs, la conception de ce matériel s'appuie sur le partage de ressources matérielles entre les différents cœurs. Ce partage crée un *couplage implicite* des applications s'exécutant en parallèle. En effet, l'accès concurrent aux ressources partagées par les différents cœurs est une source d'*interférences*, occasionnant le plus souvent des ralentissements. Nous pouvons distinguer deux types d'interférences :

- Une application souffre d'interférences *spatiales* lorsqu'une ressource est rendue indisponible par une autre application. Par exemple, des données en cache évincées au profit d'une autre application.

- Une application souffre d'interférences *temporelles* lorsque son temps d'accès à une ressource est allongé au profit d'une autre application. Par exemple, lors de l'accès à un bus partagé.

Les ressources matérielles partagées entre les cœurs sont donc autant de *canaux d'interférences* pour les applications s'y exécutant en parallèle. Un inventaire de ces différents canaux a été dressé par Kotaba et al. [54], que nous reprenons dans le tableau 2.1. Donc, du fait des hypothèses formulées dans la section précédente, nous ne nous préoccupons pas des canaux d'interférences suivants :

- *Les interférences des étages de pipelines*, car nous faisons l'hypothèse d'architectures ne disposant pas d'hyperthreading.
- *Les interférences sur les périphériques et les unités logiques*, car l'accès à ces ressources se fait à travers le logiciel d'infrastructure. Il ne s'agit donc pas à proprement parler d'une ressource accédée implicitement.
- *Les interférences dues aux protocoles de cohérences de données*, car nous faisons l'hypothèse d'un modèle de programmation asymétrique (AMP). Les données d'une application ne peuvent donc à priori pas être invalidées.

2.2.1 Impact sur nos travaux

Les interférences sont une source de ralentissements dans les processeurs multi-cœurs COTS. Elles peuvent être induites par des arbitrages d'accès (interférences temporelles) ou bien par le vol de ressources entre les différents cœurs (interférences spatiales). Elles peuvent se produire dans différents composants que nous désignons par le terme canal d'interférences. Parmi les canaux d'interférences recensés sur le matériel actuel, notre regard se porte sur ceux du système mémoire. Dans la section suivante, nous allons expliquer plus en détail les différentes interférences pouvant affecter ce dernier.

2.3 Système mémoire

Au fil des années, le système mémoire est devenu un élément de plus en plus critique dans la performance des processeurs. Une explication est donnée par la figure 2.3 illustrant l'évolution dans le temps de la performance des processeurs et de celles du temps d'accès à la mémoire. On peut y constater qu'entre 1980 et 2015, les processeurs sont devenus 10 000 fois plus rapides, tandis que le temps d'accès à la mémoire n'a été divisé que par 10. Dans ces conditions, la dégradation des performances du système

3. S : spatial T : temporel

Canal	Type [3]	Interférence
Bus système	T	contention causée par les coeurs
	T	Contention causées par les périphériques
	T	Contention causées par les procoles de cohérences
Passerelles d'interconnexion	T	Contentions causées par les autres bus
Mémoire principale	S	Fermetures de pages par d'autre applications
	T	Réordonancements de requêtes d'accès
Caches partagés	S	Éviction de donnés par d'autres applications
	T	Accès concurrent au cache
	S	Invalidation de données par les protocoles de cohérences
Caches locaux	S	Invalidation de données par les protocoles de cohérences
TLB	S	Invalidation d'entrées par les protocoles de cohérences
Périphériques	T	Contention entre applications
	T	Routage des interruptions
Étages des pipeline	T	Contention dans les architectures hyperthreadés
Unité logiques (Coprocesseurs, GPU, ...)	T	Contention entre applications.

TABLE 2.1 – Interférences rencontrées dans les cibles multicœurs (source Kotaba et al. [54])

mémoire par les interférences va avoir un impact important sur la performance globale du matériel. C'est pour cette raison que la mémoire est au cœur des préoccupations soulevées par les interférences.

Dans le reste de cette section, nous présenterons le système mémoire et les différentes interférences pouvant l'affecter. Nous commencerons par décrire les caractéristiques des mémoires actuelles et comment elles sont organisées. Nous nous pencherons ensuite sur le fonctionnement de la mémoire principale, puis sur celle des caches.

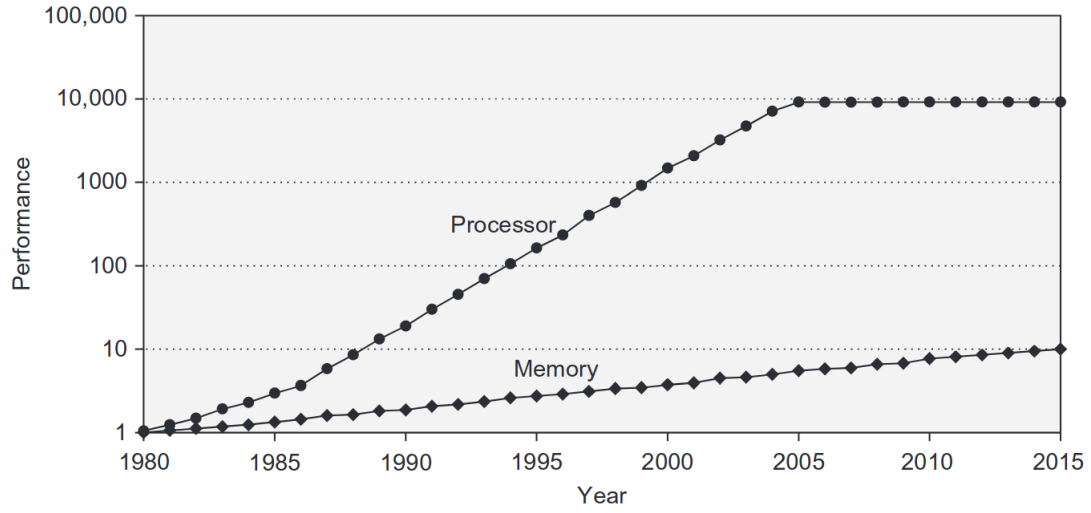


FIGURE 2.3 – Évolution des performances respectives de la mémoire et des processeurs, normalisés par leur niveau en 1980 (Source de l'illustration Hennessy et Patterson [43]).

2.3.1 Caractéristiques des mémoires et organisation

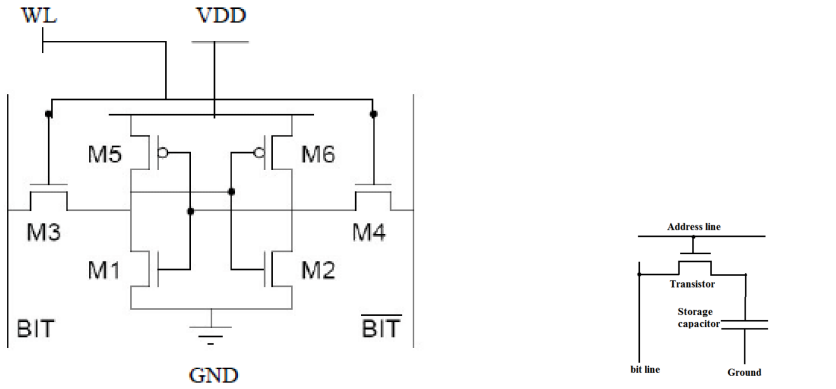
La *mémoire vive ou Random Access Memory (RAM)* désigne la mémoire dans laquelle les données peuvent être stockées et effacées. On peut la représenter comme un ensemble de *cellules* connectées entre elles, chacune stockant un *bit* de donnée. Il y a deux grandes familles de technologie pour les cellules de mémoire vive : les cellules de *mémoire statiques ou Static Random Access Memory (SRAM)* et les cellules de *mémoire dynamiques ou Dynamic Random Access Memory (DRAM)*.

La mémoire statique fonctionne sur le principe d'une bascule RS. Une bascule RS est une porte logique avec deux entrées R et S et deux sorties Q et $\bar{Q}$. Si $S = 1$ et $R = 0$, alors $Q = 1$. Si $S = 0$ et $R = 1$, alors $Q = 0$. Lorsque R et S valent 0 alors la valeur de Q est inchangée. R et S ne peuvent pas valoir 1 simultanément. On peut donc utiliser une bascule RS pour stocker un bit. Lire une mémoire statique consiste simplement à mesurer la sortie de la bascule, la rendant extrêmement rapide. La contrepartie est la faible densité d'intégration que permet cette technologie. En effet, la SRAM est le plus souvent de type 6T (illustré sur la figure 2.4b) qui utilise 6 transistors, ce qui s'avère extrêmement coûteux.

La mémoire dynamique (DRAM) stocke la valeur d'un bit dans un condensateur gardé par un transistor. Si le condensateur est chargé le bit vaut 1, sinon il vaut 0. La

mémoire est lue en fermant le transistor, et en vérifiant si le transistor était chargé ou non. Cette technologie permet une grande densité d'intégration, vu que seulement un transistor et un condensateur sont requis. Elle présente néanmoins trois inconvénients :

1. Les latences de lecture et d'écritures sont importantes, du fait qu'elles sont contraintes par le temps de charge et décharge du condensateur.
2. Vu que le condensateur doit être déchargé pour déterminer l'état de la cellule, la lecture est destructive.
3. Les condensateurs se déchargent avec le temps. L'état des cellules doit donc être rafraichi périodiquement pour éviter de perdre des données. Cela consiste simplement à lire puis réécrire les données.



(a) Cellule de SRAM 6T (b) Cellule de DRAM

FIGURE 2.4 – Circuit des différents types de cellules mémoires actuellement disponible.

Les caractéristiques de la DRAM et de la SRAM font qu'à coût équivalent, une mémoire est, à l'heure actuelle, soit rapide soit de grande capacité. La surface étant l'un des principaux facteurs de coûts d'un composant électronique. Afin de donner l'illusion d'une mémoire à la fois rapide et de grande capacité, les systèmes mémoires sont organisés hiérarchiquement ; chaque niveau de cette hiérarchie étant à la fois plus éloigné du processeur et comportant une mémoire plus grosse et plus lente que le niveau inférieur. Ce type d'architecture est très répandu, il s'agit d'ailleurs d'une idée assez ancienne, remontant au fondement des premiers ordinateurs [98]. La hiérarchie mémoire type rencontrée dans le matériel utilisé dans les terminaux mobiles est illustré dans la figure 2.5.

Le niveau le plus bas de la hiérarchie mémoire est occupé par les registres, utilisés par le processeur pour le stockage des opérandes et des résultats intermédiaires. Le

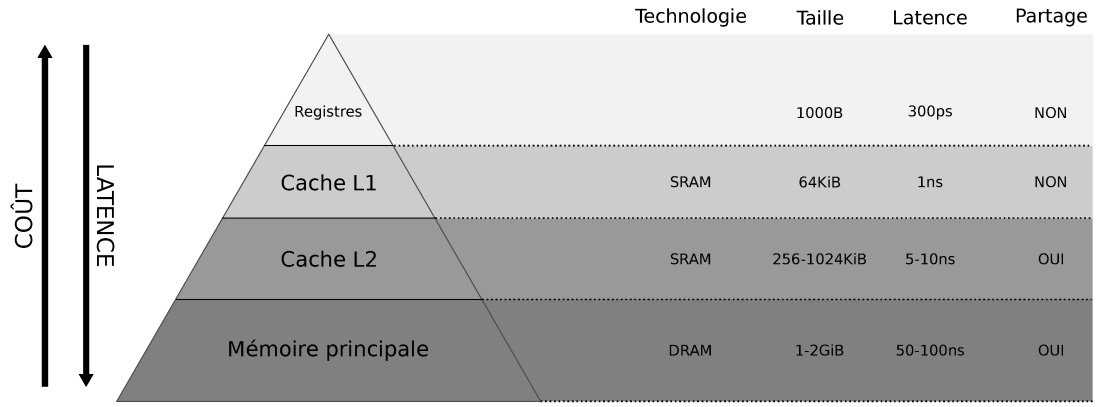


FIGURE 2.5 – Hiérarchie mémoire couramment rencontrée dans les terminaux mobiles
.

niveau le plus haut est occupé par la mémoire principale basée sur la technologie DRAM. Les niveaux intermédiaires sont occupés par les *caches*, destinés à accélérer l'accès à la mémoire principale en stockant les données accédées récemment dans de la SRAM. Cette architecture est conçue pour exploiter deux caractéristiques du comportement d'accès à la mémoire couramment observé dans les programmes : la localité spatiale [61] et la localité temporelle [101]. La propriété de localité spatiale exprime une forte probabilité que lorsqu'une donnée est accédée, les données voisines sont accédées dans un futur proche. Tandis que la propriété de localité temporelle exprime une forte probabilité qu'une donnée accédée récemment le soit de nouveau dans un futur proche. Dans les processeurs multicœurs, les niveaux supérieurs de cette hiérarchie sont généralement partagés, faisant des caches et de la mémoire principale d'éventuels canaux d'interférences. Nous allons maintenant détailler le fonctionnement et l'impact des interférences sur ces deux composants.

2.3.2 Mémoire principale

La mémoire principale est implantée par des cellules de DRAM afin d'offrir une grande capacité de stockage. L'utilisation de ce type de cellules apporte un certain nombre de contraintes, qui impose une organisation particulière que nous allons maintenant détailler.

Organisation

La mémoire principale est composée de modules (Dual Inline Memory Module ou DIMM) et de contrôleurs mémoires. Un ensemble de modules mémoire est associé à un contrôleur mémoire par le biais d'un *canal de communication ou channel*. Ce canal est constitué de trois bus : un *bus de commande*, un *bus d'adresse*, et un *bus de données*. Le contrôleur reçoit des requêtes d'accès mémoires qu'ils décomposent en une série de commandes. Les bus de commandes, d'adresses et de données sont alors utilisés respectivement pour transférer les commandes, les adresses de destination et les données lues ou écrites.

Un module mémoire comprend plusieurs *puces mémoires* qui sont regroupées en ranks. Une requête mémoire est émise à destination d'un rank en particulier, les puces qui le composent sont alors activées. Les puces constituant un rank partagent le bus de commandes et d'adresse, mais occupent chacune une portion du bus de donnée. Cela signifie que les données sont transférées en parallèle à partir de toutes les puces du rank, permettant ainsi de compenser la latence de la DRAM par une bande passante plus élevée.

Chaque puce est constitué de plusieurs *bancs mémoires* ou plus rarement *pages* (à ne pas confondre avec les pages de mémoire virtuelle). Un banc mémoire comprend une matrice de cellules DRAM organisé en ligne et en colonne, plus une ligne supplémentaire appelée *row buffer*. Lorsque une donnée est accédée dans un banc mémoire, toute la ligne à laquelle elle appartient est chargée dans le row buffer. Il y a deux raisons à cela. D'une part, cela permet de gérer le fait que lire une cellule de DRAM détruit son contenu. D'autre part, cela permet d'accélérer l'accès aux données présentes sur la même ligne. Si la ligne à laquelle appartient une donnée auquelle on souhaite accéder n'est pas chargée, il y a alors *un défaut de ligne ou row miss*. Il faut alors écrire la ligne présente dans le row buffer (qu'elle ait été modifiée ou non) avant de charger la ligne désirée.

Contrôleur DRAM

Le contrôleur DRAM est en charge de traiter les différentes requêtes d'accès à la DRAM, tout en maintenant cette dernière dans un état de fonctionnement correct. Il gère entre autres les différentes contraintes temporelles, dont le rafraichissement des cellules. Le contrôleur reçoit des requêtes de lectures et d'écritures qu'il traduit en un ensemble de commandes agissant sur les bancs mémoires. Plusieurs commandes sont définies par les JEDEC [92](l'organisation gérant les normes concernant la mémoire dynamique),

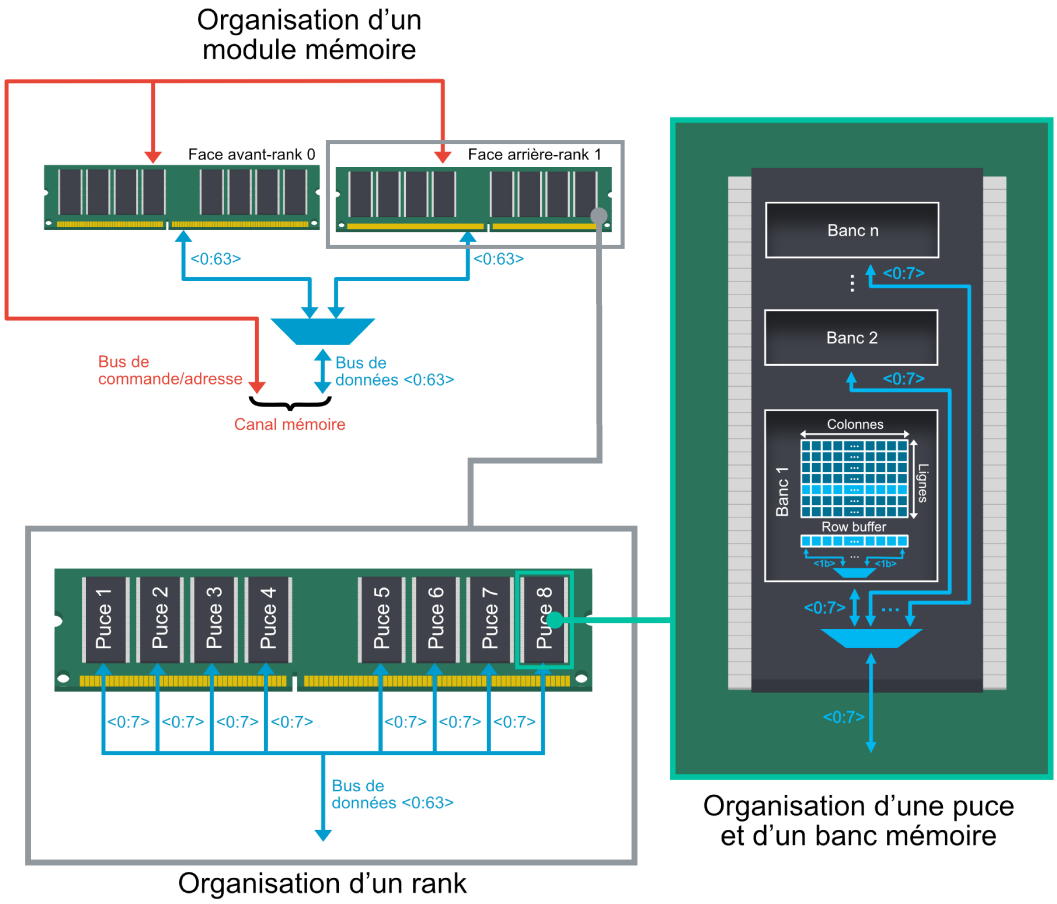


FIGURE 2.6 – Organisation de la mémoire principale

parmi lesquels nous citerons les suivantes :

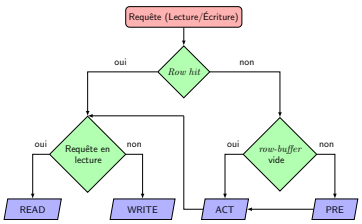
- `ACT` charge une ligne dans le row buffer.
- `PRE` Vide le contenu du row buffer dans les cellules mémoire correspondantes.
- `READ` Lecture depuis le row buffer
- `WRITE` Écriture dans le row buffer
- `REF` Rafraichissement d'une ligne. Il s'agit de la combinaison d'une commande `ACT` et `PRE`. Cela a pour conséquence de vider le row buffer.

Les row buffers jouent un rôle essentiel dans la performance dans le DRAM. Ils peuvent être gérés suivant deux politiques.

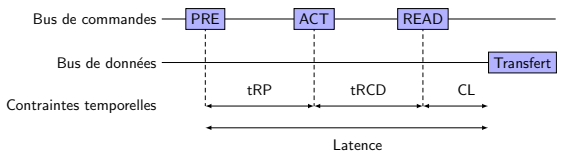
- *La politique open-row* Une ligne chargée dans un row buffer le reste jusqu'a son éviction. Cette stratégie est optimale pour les accès séquentiels. Par contre, le coût des défauts de lignes est élevé, étant donné qu'il faut écrire le contenu du row buffer avant de charger la nouvelle ligne.

- *La politique close-row* La ligne est close après chaque accès. Cette politique offre un meilleur pire cas en évitant les conflits sur les row buffers, mais dégrade les performances en moyenne.

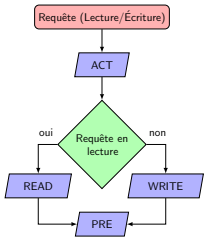
Le contrôleur mémoire est également en charge de maximiser les performances d'accès à la DRAM. Pour cela, il réordonnance les requêtes d'accès afin de maximiser le débit de requêtes traitées et la bande passante de la mémoire principale. La politique le plus couramment utilisée est *Premier Prêt/Premier Arrivé Premier Servi ou FR/FCFS*. Dans cette politique, les requêtes sont traitées dans l'ordre d'arrivée, mais une priorité est donnée à celle qui sont *prête à être traitées*. En pratique, une requête est prête à être traitée si la ligne correspondante est chargée, et si le bus de donnée est dans le bon sens (lecture ou écriture). Le contrôleur optimise donc la bande passante globale aux dépens de l'équité des requêtes.



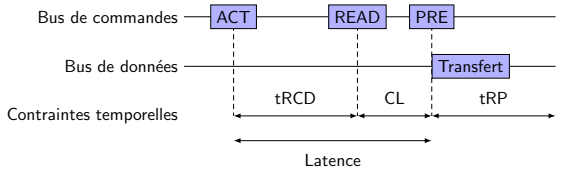
(a) Algorithme de la politique *open-row*



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



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



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

FIGURE 2.7 – Algorithmes et pire cas des politiques de gestion de row buffer

Mémoire principale et interférences

La mémoire principale est un canal d'interférences spatiales et temporelles pour les différents cœurs qui l'utilisent.

Deux types de conflits peuvent survenir. Les *conflits de bancs* surviennent lorsque plusieurs cœurs tentent d'accéder au même banc mémoire simultanément. Ce type de conflit est une interférence temporelle : il cause une séquentialisation des commandes. Les *conflits de lignes* surviennent lorsque deux commandes accèdent à deux lignes différentes sur le même banc, entrainant ainsi des rechargements de row buffer. Il s'agit ici d'une interférence spatiale affectant la disponibilité des row buffers. Ce type de conflits peut techniquement être évité en utilisant la politique close-row. Mais en pratique, c'est la politique open-row qui est employée, vu qu'elle permet d'atteindre de meilleures performances en moyenne.

La politique de réordonnancement de requête est une autre source d'interférence temporelle. La politique FR/FCFS donne en effet la priorité aux requêtes d'accès pouvant être traitée immédiatement. En conséquence, un cœur utilisant la DRAM efficacement, c'est-à-dire jouissant d'une bonne localité spatiale et temporelle, sera plus prioritaire qu'un cœur avec une utilisation moins efficace. On peut en conclure que l'iniquité de cette politique rend plus sensibles aux interférences sur la mémoire principale les applications l'utilisant de façon non optimale.

2.3.3 Caches

Les caches ont pour but de réduire la latence moyenne d'accès à la mémoire. Pour cela, ils stockent un sous-ensemble des données présentes en mémoire dans une mémoire SRAM, qui est donc très rapide, mais de taille limitée. En stockant les données accédées récemment, ils permettent de considérablement améliorer les performances des programmes présentant une bonne localité spatiale et temporelle. En effet, comme on peut le constater dans la figure 2.5, la latence d'accès à la mémoire principale peut atteindre l'ordre de la centaine de nanosecondes contre une nanoseconde pour un accès au cache de niveau un. Les caches sont donc un élément crucial pour la performance des processeurs modernes.

Accès

L'accès au cache est complètement transparent pour le programmeur. Lorsqu'une donnée est accédée, le cache de plus bas niveau est d'abord interrogé sur sa présence. Si

la donnée est présente dans le cache, elle est alors accédée en lecture ou écriture depuis celui-ci. Dans le cas contraire, la donnée doit alors être chargée depuis le niveau suivant de la hiérarchie mémoire, on parle alors de *défaut de cache ou cache miss*. Le niveau suivant de la hiérarchie mémoire peut alors être directement la mémoire principale, ou bien un autre cache. Nous pouvons distinguer quatre types de défauts de caches :

- *Défauts obligatoires* La donnée est chargée pour la première fois.
- *Défauts capacitifs* Le cache est plein, la donnée doit alors être chargée aux dépens d'une autre.
- *Défauts conflictuels* deux données distinctes partagent le même emplacement de cache et s'évincent mutuellement.
- *Défauts de cohérence* La donnée a été invalidée par un protocole de cohérence.

Organisation

Un cache est divisé en *blocs* appelés *lignes*. Une ligne de cache comprend plusieurs mots mémoires (généralement huit). Les transferts de données se font à la granularité du mot mémoire vers le processeur, et à la granularité de la ligne de cache vers les autres niveaux de la hiérarchie mémoire. Des métadonnées sont associées à chaque ligne de cache, leur nature exacte pouvant varier légèrement d'un type de cache à l'autre.

Le placement des données dans le cache se fait au moyen de l'adresse de la donnée accédée. Cette dernière contient trois informations essentielles :

- *L'index* désigne un sous-ensemble de lignes de caches pouvant accueillir la donnée.
- *L'offset* désigne la position d'un octet dans la ligne de cache.
- *Le tag* permet de déterminer l'adresse de la donnée présente dans une ligne.

La position des bits encodant cette information est décrite dans la figure 2.8. Le découpage choisi n'est pas anodin. Il associe aux lignes adjacentes en mémoire des index adjacents dans le cache, prévenant ainsi des défauts conflictuels pour les applications jouissant d'une bonne localité spatiale.

On peut utiliser aussi bien l'adresse virtuelle que l'adresse physique pour tagger et indexer le cache. On rencontre trois cas :

- *Cache Virtuellement Indexé et Virtuellement Taggé (VIVT)* Seule l'adresse virtuelle est utilisée pour interroger le cache. Cette méthode a l'avantage de la rapidité, vu qu'elle ne nécessite pas de traduire l'adresse. Elle pose néanmoins le problème

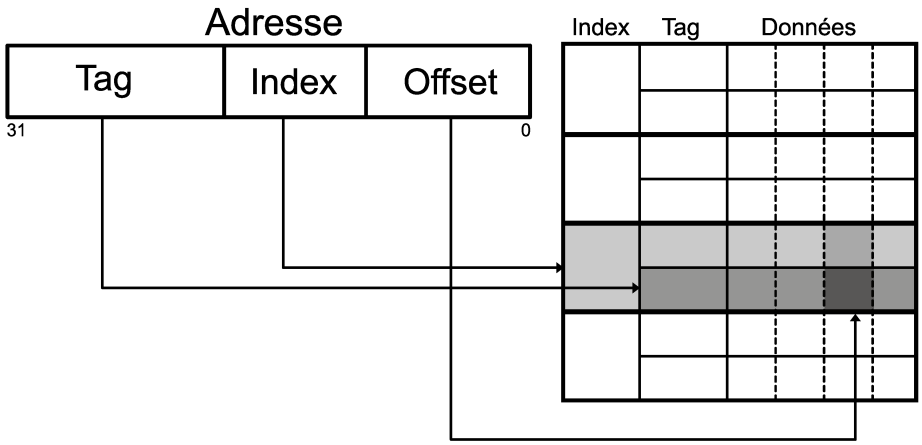


FIGURE 2.8 – Correspondance entre les bits d'adresses et le placement des données dans un cache

des homonymes, deux mêmes adresses virtuelles pouvant désigner deux emplacements physiques différents. Ce problème a d'autant plus de chance de survenir dans le cas où le cache est partagé. Si des solutions à ce problème existent, elles se révèlent incompatibles avec les protocoles de cohérences de caches classiques [49].

- *Cache Physiquement Indexé et Physiquement Taggé (PIPT)* Seule l'adresse physique est utilisée. Cette méthode permet d'éviter le problème des homonymes, mais présente l'inconvénient de la lenteur, une traduction de l'adresse étant nécessaire pour interroger le cache.
- *Cache Virtuellement Indexé et Physiquement Taggé (VIPT)* Cette approche vise à réduire le temps d'accès au cache en accédant à la ligne à tester pendant la traduction de l'adresse. L'utilisation de l'adresse physique permet d'éviter les problèmes d'homonymies. Pour cela, il faut néanmoins qu'il n'y ait aucun bit d'index ne soit traduit, ce qui limite cette approche aux caches de petite taille.

Politiques de correspondance

La politique de correspondance d'un cache définit quelle ligne peut être occupée pour un index donné.

Si un index ne désigne qu'une seule ligne dans le cache, celui-ci est dit à *correspondance préétablie (direct mapped)* (figure 2.9a). Cette politique a l'avantage d'être peu coûteuse à

mettre en œuvre. En effet, lorsque le cache est interrogé, il suffit de tester une seule ligne. Elle a par contre l'inconvénient d'être à l'origine de nombreux défauts conflictuels.

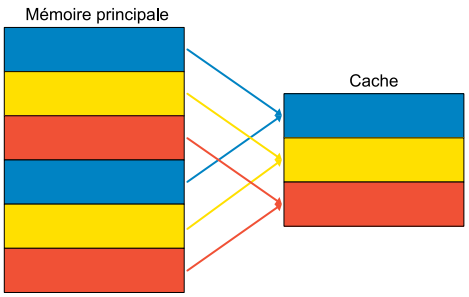
La politique réciproque est de faire correspondre n'importe quelle ligne du cache pour un index donné. Un cache utilisant cette politique est alors dit *pleinement associatif ou full associative* (figure 2.9b). Lors des défauts capacitaires, une *politique de remplacement* est appliquée afin de déterminer quelle ligne va être remplacée. Différentes politiques de remplacement peuvent être appliquées : *FIFO*, *aléatoire*, *Least Recently Used*. Si cette politique de correspondance offre une flexibilité optimale à la politique de remplacement, elle nécessite également de tester toutes les lignes de caches lorsque le cache est interrogé, ce qui se traduit par des coûts d'implantations prohibitifs.

La politique de correspondance utilisée en pratique est *l'association partielle ou set associativity* (figure 2.9c). Il s'agit d'un compromis entre la correspondance préétablie et la pleine association. Cette politique associe à chaque index un ensemble de taille fixe de lignes. En cas de défaut capacitif, la politique de remplacement choisit la ligne à remplacer parmi les lignes de l'ensemble correspondant. Le cache est alors divisé en *voies ou cache ways* correspondant aux différents emplacements d'un ensemble de lignes. *L'associativité* désigne la taille des ensembles de lignes, et donc le nombre de voies du cache. On peut voir un cache à correspondance préétablie comme un cache partiellement associatif avec une seule voie. De même, un cache pleinement associatif peut être vu comme un cache partiellement associatif ne comportant qu'un seul ensemble de lignes.

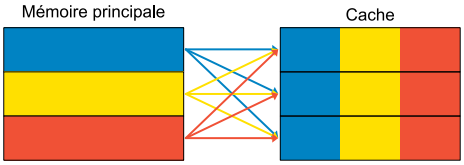
Politiques d'écritures

La politique d'écriture d'un cache détermine quand les données sont écrites vers le niveau supérieur. On en dénombre deux :

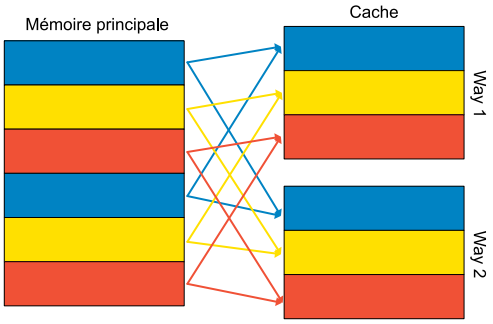
- *Écriture directe (Write Through)* Les écritures sont immédiatement propagées vers le niveau supérieur. Cette politique a l'avantage de simplifier la maintenance de la cohérence entre les différents niveaux de la hiérarchie mémoire ; au prix, néanmoins d'une plus grande consommation de bande passante.
- *Écritures différées (Write Back)* Les données ne sont écrites vers le niveau supérieur que lors de leur remplacement. Les métadonnées des lignes des caches utilisant cette politique comprennent un bit *dirty*, qui est activé lorsque les données de la ligne sont modifiées. Lors du remplacement d'une ligne, le bit dirty est d'abord testé. La donnée n'est alors écrite en mémoire que dans le cas où ce dernier a été activé. Cette politique permet de réduire le trafic vers les niveaux supérieurs de



(a) Association directe



(b) Associativité totale



(c) Associativité partielle

FIGURE 2.9 – Politiques de correspondances

la hiérarchie mémoire, mais elle complexifie également la gestion de la cohérence des données.

Politiques d'allocation

La politique d'allocation détermine si une donnée chargée depuis le niveau de cache supérieur doit être copiée dans le cache (*line fill*) ou non. Il y a deux politiques d'allocations :

- *Read-allocate* Un line fill n'est effectué que lors des défauts de cache en lecture. Lors d'un défaut de cache en écriture, le cache n'est pas modifié et l'écriture transmise au niveau supérieur.

- *La politique write-allocate* Un line fill a lieu lors des défauts de caches en lecture et en écriture. Cette politique est généralement utilisée dans les caches *write-back*.

Caches et interférences

Les caches partagés sont des canaux d'interférences importants dans les processeurs modernes. Elles peuvent être aussi bien spatiales que temporelles. Elles entrainent trois effets indésirables majeurs :

- *Interférence temporelle lors de l'accès au cache* Les caches ont la capacité de traiter plusieurs accès en parallèle. Cette capacité peut néanmoins être excédée lors de l'utilisation en simultané du cache par plusieurs cœurs. Dans ce cas, des requêtes ayant dû être traitées en parallèle le sont séquentiellement induisant une latence supplémentaire.
- *Contention inter-cœurs sur les lignes de cache* Les lignes de caches d'une application s'exécutant sur un cœur peuvent être remplacée au profit des lignes d'une application sur un autre cœur, ce qui a pour conséquence une augmentation des défauts conflictuels. Ces interférences spatiales, dites inter-cœur, ont un impact considérable sur les pires temps d'exécution, se traduisant par un surdimensionnent inacceptable du matériel.
- *Interférence spatiale causée par les protocoles de cohérences* Lorsque des données sont partagées entre les cœurs, il faut maintenir la cohérence entre les différents caches du système. Les protocoles utilisés pour maintenir cette cohérence peuvent *invalider* des données, provoquant ainsi une augmentation des défauts de cohérences. Ce problème touche surtout les systèmes adoptant le modèle symétrique.

2.4 Conclusions du chapitre

Dans cette section, nous avons présenté le problème des interférences dans les processeurs multi-cœurs destinés au marché de masse. Ces interférences, dues aux accès concurrents aux ressources partagés entre les cœurs, sont la source de ralentissement pour des applications, empêchant ainsi l'isolation temporelle requise pour garantir la sûreté des systèmes intégrés temps-réel. Ce problème concerne notamment le système mémoire, qui est à ce jour l'un des principaux acteurs de la performance des processeurs modernes.

Les interférences sont un problème, car elles peuvent causer la défaillance des applications temps-réel du système. Dans le chapitre suivant, nous allons nous pencher plus en détail sur les aspects temps-réel et l'impact des interférences sur ces derniers. Nous y présenterons également les approches proposées par la communauté scientifique pour apporter une solution au problème des interférences.

CHAPITRE 3

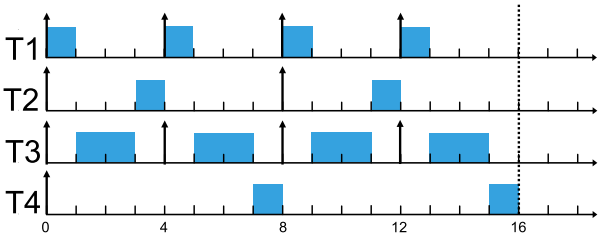
Temps-réel et interférences

Dans le chapitre précédent, nous avons vu que les processeurs multi-cœurs COTS présentent le problème des interférences dues au partage de matériel entre les cœurs. Ces interférences étant à l'origine de ralentissements pour les applications s'exécutant en parallèle, elles posent particulièrement problème dans les systèmes temps-réel. Ce chapitre expose les problématiques liées à l'utilisation de processeurs multi-cœur COTS dans ce type de système.

Ce chapitre est organisé de la façon suivante. Dans la première section, on expose des généralités sur les systèmes temps-réels, notamment sur les contraintes que ceux-ci doivent respecter. Dans la deuxième section, nous présentons les différentes approches employées pour s'assurer du respect de ces contraintes. Dans la troisième section, nous exposons les conséquences qu'ont les interférences dans la mise en œuvre de ces méthodes d'analyses. Enfin, dans la quatrième et dernière section de ce chapitre, nous faisons un état de l'art des approches proposées pour la gestion du problème des interférences dans les systèmes temps-réels.

3.1 Généralités sur les systèmes temps-réels

Un *système temps-réel* est un système comprenant des tâches devant respecter des *échéances*. Cela ne signifie pas que le temps de réaction doit être rapide, mais qu'il doit être *borné*. Le modèle couramment utilisé, introduit par Liu et Layland [63], définit un système temps-réel comme un ensemble de *tâches périodiques*. Une tâche y est caractérisée par une période, une échéance, et une capacité qui est le temps nécessaire à l'exécution



	T1	T2	T3	T4
Période	4	8	4	16
Capacité	1	1	2	2
Échéance	4	8	4	16

FIGURE 3.1 – Exemple d'ordonnancement obtenu avec la politique *Earliest Deadline First*.

de la tâche. Chaque tâche est activée au début de sa période et doit se terminer avant son échéance. Si l'échéance est dépassée, cela se traduit soit par une invalidation du résultat de la tâche (on parle alors de *temps-réel dur*), soit par une dégradation de la qualité de service du système (on parle de *temps-réel mou*).

Un des enjeux majeurs de la conception d'un système temps-réel est d'en garantir la *sûreté temporelle*, c'est-à-dire s'assurer que toutes les tâches du système respectent leurs échéances. Le procédé usuel consiste en deux analyses successives.

1. *L'analyse de pire temps d'exécution* dont le but est de dimensionner les capacités des tâches du système avec le plus long temps d'exécution possible.

2. *L'analyse d'ordonnancement* dont le but est de déterminer le temps de réponse des tâches du système pour une politique d'ordonnancement donnée.

Un système est dit *ordonancable* si on peut montrer que le temps de réponse de toutes les tâches du système est inférieur ou égal à son échéance. Un exemple de système ordonancable et son ordonnancement est donné figure 3.1.

Ce processus repose sur une hypothèse forte, qui est que le pire temps d'exécution d'une tâche est indépendant des autres tâches du système. En d'autres termes, il repose sur la composabilité du système. Or, nous avons vu que cette dernière n'est pas assurée en présence d'interférences. Le choix du matériel a donc un impact important sur l'analyse de pire temps d'exécution. C'est ce problème que nous allons étudier dans le reste de ce chapitre.

3.2 Analyse de pire temps d'exécution en mono-cœur

Dans un système temps-réel, on suppose que la capacité des tâches correspond au temps nécessaire au parcours de leur plus long chemin d'exécution [1] sur une plateforme matérielle donnée. Ce temps est désigné par le terme *WCET* [2](figure 3.2). Le but de l'analyse de pire temps d'exécution est de borner le WCET. Cette analyse se doit d'être *sûre* et *précise* : le WCET estimé ne doit pas être inférieur au WCET réel (sûreté) et il doit s'en approcher autant que possible (précision).

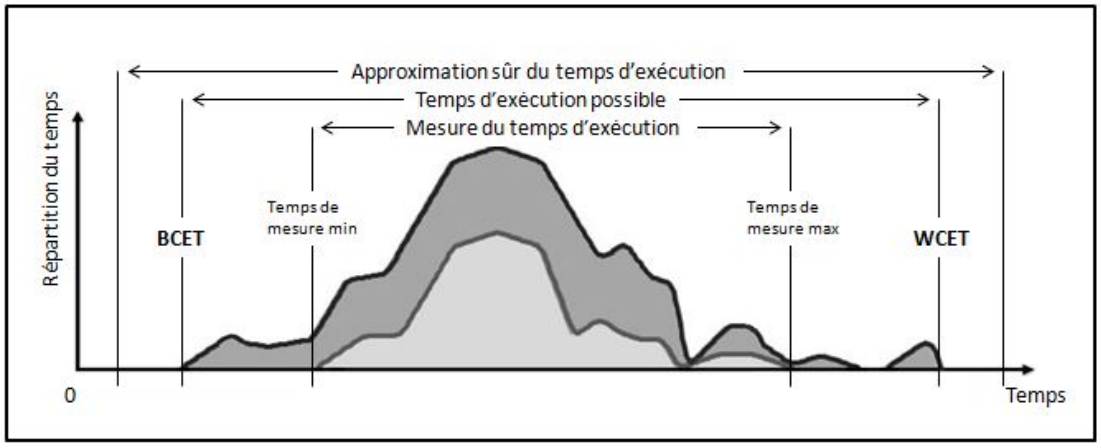


FIGURE 3.2 – Résumé des notions concernant l'analyse de pire temps d'exécution. La courbe la plus haute représente l'ensemble de toutes les exécutions possibles, tandis que la courbe la plus basse représente un sous-ensemble d'exécution mesurée. (Source de l'illustration : Wikipedia)

Nous allons maintenant détailler deux grandes familles d'analyse du pire temps d'exécution : celles basées sur l'analyse statique de l'application (on parle de méthodes statiques) et celles basées sur des mesures (on parle de méthodes dynamiques).

3.2.1 Méthodes statiques

Les analyseurs statiques [32, 62, 93, 19, 60] de WCET le calculent sans exécuter directement les tâches. Une architecture générale de l'analyse employée par ces outils est décrite par Wilhelm et al. [100] et résumée par la figure 3.3.

Les points d'entrée de ce type d'analyse sont le fichier binaire exécutable de la tâche analysée et un ensemble d'informations complémentaires sous forme d'annotations.

1. La longueur est ici exprimée en temps
2. Worst Case Execution Time

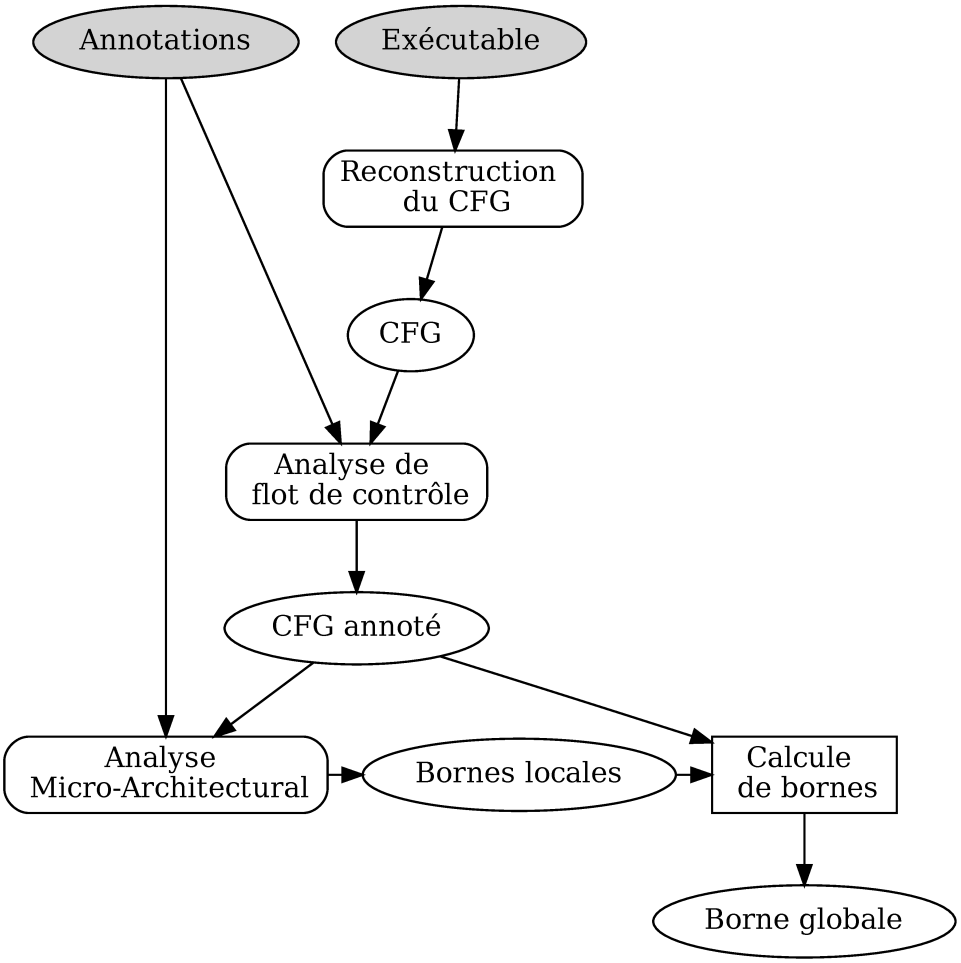


FIGURE 3.3 – Analyse statique de pire temps d'exécution

Ces informations peuvent décrire l'agencement de la mémoire, l'intervalle de valeur d'entrées de la tâche, des informations sur son flot de contrôle, etc. Le processus de calcul de WCET à partir de ces éléments est le suivant :

1. *Reconstruction du graphe de flots de contrôle* Le flot de contrôle est reconstruit à partir du binaire afin d'extraire une représentation intermédiaire pour la suite de l'analyse. Cette étape est comparable à un frontend de compilateur. Le produit de cette étape est donc un *graphe de flot de contrôle (CFG)*.

2. *Analyse du flot de contrôle* Le CFG est analysé afin de construire l'ensemble des chemins d'exécutions pouvant être possiblement être emprunté par la tâche. Un des objectifs de cette étape est d'éliminer de façon sûre le plus de chemins possible. Le produit de cette étape est un CFG annoté avec des contraintes sur le comportement dynamique de la tâche.
3. *Analyse micro architecturale* Une borne sur le temps d'exécution des blocs de bases du CFG sont calculés en utilisant un modèle abstrait du matériel. Le produit de cette étape est un ensemble de *bornes locales*.
4. *Calcul de la borne global* Finalement, les bornes locales et les informations sur le flot de contrôle sont utilisées pour déterminer la *borne globale* sur le pire temps d'exécution de la tâche.

3.2.2 Méthodes dynamiques

L'analyse dynamique de pire temps d'exécution utilise des expériences pour déterminer le pire temps d'exécution d'une tâche. Elle peut s'effectuer aussi bien de bout en bout que par morceau. Dans l'analyse de bout en bout, c'est le temps complet d'une exécution de la tâche qui est mesuré. Tandis que dans l'analyse par morceau, les mesures sont effectuées sur les blocs de bases de l'application.

Qu'elles soient de bout en bout ou par morceaux, l'analyse dynamique de pire temps d'exécution est généralement considérée comme non sûre. D'une part, il faut s'assurer que le chemin correspondant au WCET est bien couvert par les mesures, ce qui n'est en général pas possible avec les processeurs modernes. Ainsi, si l'on se réfère à la terminologie de la figure 3.2, l'analyse de bout en bout ne permet techniquement d'obtenir *qu'un pire temps d'exécution observé*. Elles peuvent néanmoins être utilisées pour étudier la variabilité des temps d'exécution d'une tâche, ou encore pour valider les résultats de l'analyse statique.

3.2.3 Défis de l'analyse de pire temps d'exécution

Qu'elle soit statique ou dynamique, la complexité de mise en œuvre de l'analyse de pire temps d'exécution est fortement liée à la complexité du matériel.

Le chemin d'exécution avec le WCET dépend à la fois des données en entrée de la tâche et de l'état dans lequel se trouve le matériel. Ces deux données étant en général difficiles à déterminer à l'avance. Plus le matériel est complexe, plus le nombre d'états dans lequel celui peut se trouver (et donc le nombre de cas à considérer) est grand. Cela

entraine des problèmes d'explosion combinatoire concernant aussi bien les méthodes statiques (il faut considérer plus de cas dans l'analyse) que les méthodes dynamiques (la couverture de tests à fournir est plus grande).

De plus, les processeurs modernes posent le problème du temps d'exécution des instructions. En effet, ce dernier n'y est pas fixe. Il dépend de l'historique des instructions exécutées précédemment. D'une part, cela rend difficilement applicable l'analyse dynamique par morceau sur ce type de matériel. D'autre part, cela complique l'analyse micro architecturale dans les approches statiques. En effet, le modèle abstrait du matériel doit alors prendre en compte l'effet de composants tel que les pipelines ou les caches pour déterminer les bornes locales de la tâche. Cela requiert un niveau d'information élevé sur le matériel qui n'est pas toujours disponible, à plus forte raison sur du matériel grand public. De plus, le comportement dans le pire cas des composants peut conduire à des bornes très grandes.

Enfin, le dernier problème qui se pose est la présence d'anomalies temporelles sur le matériel récent. Une anomalie temporelle survient lorsque le pire temps global n'est pas composés uniquement de pires temps locaux. La figure 3.4 illustre un exemple d'anomalie causé par le réordonnancement des instructions au sein du pipeline d'un processeur. On peut y voir deux ordonnancements d'instructions partiellement dépendantes entre elles. Les séquences sont identiques à l'exception de l'instruction A qui met plus de temps à s'exécuter dans le premier cas que dans le second. L'anomalie est que c'est ce dernier cas qui met le plus de temps à s'exécuter au final. Si on ne peut pas borner *par une constante* la différence de temps d'exécution causée par les anomalies temporelles, on parle alors d'*effet domino*. Les processeurs modernes présentent en général à la fois des anomalies temporelles et des effets dominos. La présence d'anomalie temporelle a plusieurs conséquences :

- Il n'est pas sûr d'utiliser l'approche gloutonne pour déterminer le chemin d'exécution correspondant au WCET.
- On ne peut pas identifier un pire état initial, permettant par exemple de procéder de façon sûre à de l'analyse dynamique par morceau.
- Une architecture matérielle présentant des anomalies temporelles et des effets dominos est dite *non-compositionelle* [40]. Une architecture est compositionnelle si on peut décomposer le pire temps d'exécution totale en la somme des contributions au pire temps d'exécution des éléments du système. La conséquence est que l'on ne peut pas modulariser aisément l'analyse de ce genre de matériel.

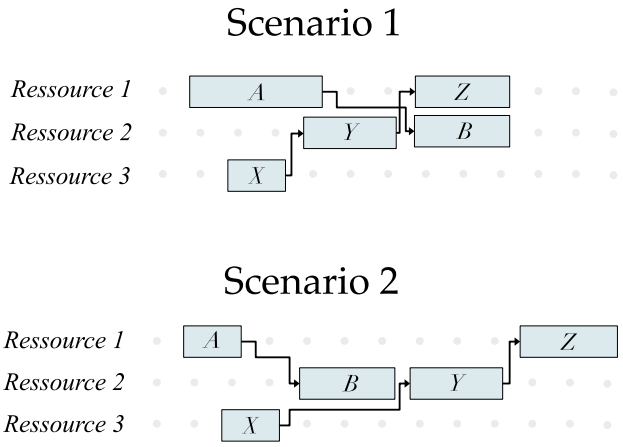


FIGURE 3.4 – Exemple d'anomalie temporelle due à un réordonnancement d'instructions. Les flèches indiquent des dépendances entre instructions.

3.3 Impact des interférences

L'utilisation de matériel multi-cœur COTS a un impact très important dans le cadre des systèmes temps-réels. Elle remet notamment en question l'applicabilité des méthodes existantes pour vérifier la sûreté temporelle de ces systèmes.

Les interférences peuvent causer des ralentissements plus ou moins importants aux tâches temps-réels, qui s' ils ne sont pas pris en compte peuvent causer un sous-dimensionnement du système entrainant des dépassements d'échéances. Le ralentissement subi par une tâche dépend à la fois de son utilisation du système mémoire, mais aussi de la pression exercée sur celui-ci par les autres tâches du système. Dans ces conditions, il devient difficile d'évaluer le pire temps d'exécution indépendamment du contexte d'ordonnancement, remettant ainsi en cause l'approche en deux étapes utilisées dans les systèmes mono-cœurs. Notons également que dans des systèmes à criticité mixte, le comportement exact des tâches non critique n'est en général pas connu, empêchant ainsi l'évaluation du contexte d'ordonnancement.

La nature du matériel utilisé pose également problème. Tout d'abord, les processeurs multi-cœurs COTS sont en général très complexes et leur comportement n'est en général pas documenté exhaustivement. Dans ces conditions, construire un modèle abstrait d'un tel processeur est difficile et peut conduire à l'utilisation d'hypothèses simplificatrices

souvent très pessimistes. Ensuite, ces processeurs sont conçus pour offrir de bonnes performances en moyenne, et donc le comportement des mécanismes utilisés peut être très mauvais dans les pires cas. Enfin, la non-compositionnalité des architectures employées aggrave encore ce problème, dans la mesure où elle complique la modularisation de l'analyse.

Dans les faits, les points que nous venons de soulever entrainent l'adoption d'hypothèse très pessimistes, et donc le surdimensionnement des systèmes. D'un point de vue industriel, tout l'enjeu est de pouvoir dimensionner de façon sure, mais aussi efficace, les systèmes temps-réel en tenant compte des interférences. Dans la section suivante, nous allons présenter l'état de l'art des interférences.

3.4 Temps-réel et interférences

La présence d'interférences dans le matériel moderne fait partie des préoccupations de la communauté temps-réel depuis la fin des années 2000. Nous allons, dans cette section, présenter diverses approches proposées pour apporter des solutions à ce problème. Nous distinguerons trois grandes familles d'approches :

- Les approches traitant du *dimensionnement* du système en prenant en compte l'impact des interférences. Cela passe par l'extension des méthodes classiques de vérification temporelles, mais également par des études empiriques.
- Les approches traitant de *l'isolation* des applications ont pour but de réduire, voir supprimer, les interférences. Elles peuvent aussi permettre un dimensionnement moins pessimiste des systèmes.
- Les approches traitant de *la régulation* ont pour but de fournir un filet de sécurité à l'exécution. Ces approches reposent sur un monitorage de l'état du système afin de détecter des situations pouvant causer des défaillances, et le cas échéant prendre des mesures appropriées.

3.4.1 Calcul de pire temps d'exécution

Un grand nombre de travaux visent à incorporer l'impact des interférences dans les analyses temporelles classiques. L'enjeu de ces travaux est d'offrir des analyses à la fois sûres et efficaces : sûre, car l'impact des interférences ne doit pas y être sous-estimé, efficace, car il doit être aussi proche que possible de la réalité. Il s'agit d'un problème difficile pour les raisons invoquées dans la section 3.3. Le problème est d'ailleurs encore

ouvert. Pour un inventaire récent et détaillé des recherches dans ce domaine, nous orientons le lecteur vers l'étude de Maiza et al. [64] en complément de ce chapitre.

Les travaux étendant la vérification des propriétés temporelles des systèmes temps-réels se distinguent à la fois par le choix de l'analyse étendue (de pire temps d'exécution ou d'ordonnancement) et la modélisation des interférences.

Extension de l'analyse de pire temps d'exécution Peu de travaux incorporent les interférences dans l'analyse de pire temps d'exécution. Cela peut s'expliquer par l'absence de contexte d'exécution, rendant difficile la modélisation des interférences. On peut néanmoins, prendre en compte les interférences dans cette analyse en suivant deux approches :

- L'*approche de Murphy* [8] [3] consiste à considérer toutes les interférences possibles pour chaque accès. Pour pouvoir appliquer cette approche, le pire surcoût possible par accès doit être borné. Ce n'est pas toujours possible, et dans le cas où on pourrait le faire, les bornes calculées sont en général très pessimistes [78, 73].
- Les *approches complètement intégrées* visent à analyser simultanément toutes les tâches s'exécutant en parallèle. Ce type d'analyse offre théoriquement une précision idéale, le surcoût temporel estimé étant déterminé à partir de tous les entrelacements d'accès possible sur les ressources partagées. En pratique, l'applicabilité de ce type d'approche reste incertaine. En cause le niveau d'information requis sur toutes les tâches du système, mais aussi la complexité du calcul.

Extension de l'analyse d'ordonnancement Les interférences peuvent également être intégrées dans le calcul du pire temps de réponse. C'est-à-dire pendant l'analyse d'ordonnancement. Cette approche permet de prendre en compte le contexte d'ordonnancement, en particulier la pression éventuelle sur les ressources partagées. Néanmoins, lorsque ce contexte n'est pas disponible, par exemple dans le cas où le comportement des autres tâches n'est pas connu, des hypothèses pessimistes de consommation peuvent être employées. Un certain nombre de travaux repose sur l'utilisation de *courbes d'arrivée* comme abstraction de la consommation de ressources partagées. Ces courbes sont des fonctions associant à chaque intervalle de temps le nombre d'occurrences d'un événement. Elles permettent donc de borner quantitativement la consommation de ressource mémoire d'une tâche. Des travaux récents [74] montrent que ces courbes peuvent être déterminées par des méthodes d'analyse statiques.

3. Nommée d'après la loi éponyme.

Modèles d'interférences des composants du système mémoire L'estimation de l'impact des interférences, quel que soit la méthode employée, repose sur un modèle plus ou moins détaillé du système mémoire. Les composants étudiés dans la littérature sont les bus d'interconnexions, les caches, et la DRAM.

La principale difficulté concernant les bus d'interconnexion est le manque d'information sur la politique d'arbitrage utilisée en cas d'accès concurrents. Notons que certaines politiques d'arbitrage n'ont pas de temps de réponse borné. Les articles traitant des bus font en général des hypothèses sur cette politique, en considérant par exemple une politique TDMA. Cela pose problème pour appliquer ces résultats lorsque l'on cible du matériel COTS.

Les caches ont également fait l'objet d'une attention particulière. La majorité des travaux sont consacrés aux interférences spatiales, et donc à déterminer le nombre de défauts de caches supplémentaires induit par les interférences. Ces approches ont d'abord été limitées aux caches d'instructions [104, 109, 41], puis étendues aux caches de données [59]. Si dans les caches partagés, les interférences spatiales les plus évidentes, il ne faut pas oublier que les interférences temporelles existent. Valsan et al. [96] ont montrés ce problème sur diverses architectures. Mesurer précisément l'impact de ces interférences temporelles semble néanmoins difficile.

De multiples conceptions de contrôleur DRAM déterministe ont été proposées [10, 82, 55], mais peu d'études ont été effectuées sur les contrôleurs utilisés dans les plateformes COTS. Wu et al. [103] ont réalisé une des premières études de pire temps de réponse de la DRAM. Les auteurs y font néanmoins l'hypothèse de l'absence de politique de réordonnancement, ces dernières pouvant conduire à des temps d'accès non bornés. Cette hypothèse a ensuite été levée par Kim et al. [51, 52]. Les résultats de ces analyses restent pessimistes, et peuvent être surtout invalidés en fonction des politiques d'arbitrage effectivement implantées dans les contrôleurs.

La présence d'anomalies temporelles et d'effets dominos dans les architectures modernes pose également le problème de la compositionnalité des analyses d'interférences. En effet, rien ne dit que la somme des retards dus aux interférences de chaque composant est supérieure ou égale au retard finalement observé. Des approches pour prendre ce phénomène en compte ont été proposées par Hahn et al. [39], augmentant encore le pessimisme des analyses. Réciproquement, la capacité de recouvrement des retards offerts par les processeurs modernes par le biais de mécanismes comme les pipelines et l'exécution dans le désordre ne sont pas pris en compte.

Conclusions

Les approches présentées dans cette section visent à borner de façon sûre l'effet des interférences. Elles se heurtent néanmoins à la complexité et à l'opacité du matériel étudié. Ainsi, des hypothèses simplificatrices sont souvent employées, remettant en cause l'applicabilité de ces travaux pour des cibles COTS. Enfin, les bornes dérivées sont très pessimistes, au point de les rendre inutiles en pratique, le surcoût calculé compensant complètement le bénéfice apporté par le fait d'avoir plusieurs cœurs.

3.4.2 Approches empiriques

L'analyse statique du pire temps d'exécution est un problème difficile, dont la complexité de mise en œuvre croît avec celle du matériel ciblé. Les processeurs évoluant plus vite que les approches d'analyse statiques, ces dernières sont aujourd'hui difficilement applicables au matériel moderne. Cette situation explique un regain d'intérêt pour les approches empiriques, certes non sûres, mais considérablement plus aisées à mettre en œuvre.

Les approches empiriques ont notamment été utilisées dans un certain nombre d'études afin d'étudier l'ampleur des ralentissements que peuvent causer les interférences sur du matériel COTS. [113, 22, 81, 33, 72]. Ces études utilisent des micro-benchmarks spécifiquement conçus pour stresser les composants partagés du système mémoire, on parle de *charges*. En comparant des temps d'exécution en isolation et face à des charges, on peut évaluer les retards observés en pratique sur du matériel complexe. Ce type d'analyse n'est en principe pas sûr, vu que l'on ne peut savoir si le pire cas a effectivement été couvert. En pratique, le comportement des charges est pessimiste, et s'apparente à une tentative de déni de service sur la mémoire. Il est néanmoins souhaitable d'avoir des charges générant le plus d'interférences possible. Or, les charges générant le plus de perturbations varient d'une cible matérielle à l'autre, nécessitant un effort de conception au cas par cas. Afin de réduire cet effort, une approche automatique a été proposée par Iorga et al. [47] pour concevoir de telles charges.

Un autre usage des méthodes empiriques est de caractériser la sensibilité d'une application aux problèmes des interférences, principalement à des fins de dimensionnement. On observe deux types de caractérisation :

- *Les approches a posteriori* caractérisent la sensibilité des applications en les soumettant à de l'injection de charges.

- *Les approches a priori* caractérisent la sensibilité des applications en fonction de leur comportement en isolation.

Dans les approches a posteriori, le comportement des applications est caractérisé à l'aide de l'injection de charges. Ainsi, la sensibilité d'une application aux interférences est décrite par une courbe décrivant l'évolution du retard subi en fonction de la charge observée dans le système. Ce type d'approche a notamment été utilisée pour la colocalisation d'applications dans les centres de données [68, 23, 35, 112, 111]. Le but étant de constituer des groupes d'applications en minimisant les interférences.

Les approches a priori visent à inférer le retard que peut subir une application en fonction de son comportement en isolation. Griffin et al. [37] utilisent des réseaux de neurones profonds [58] pour cette inférence. Le comportement en isolation est caractérisé en effectuant une analyse de composantes principales [102] sur des données collectées à l'aide de compteurs de performances. Cette approche a deux inconvénients. Tout d'abord, la caractérisation reposant sur tout les événements mesurés, un grand nombre d'expériences est nécessaire. Ensuite, en fonction des événements observable sur la machine, l'efficacité de la caractérisation peut varier fortement.

Conclusions

Les approches empiriques ont l'avantage d'être immédiatement applicables. Elles reposent majoritairement sur de l'injection de charges pour déterminer les retards subis en pratique par des applications. Ces méthodes sont théoriquement non sûres. Il est en effet très difficile, sinon impossible, de s'assurer que le stress apporté par les charges correspond effectivement au pire scénario d'interférences.

3.4.3 Isolation du matériel

Nous venons d'évoquer différentes analyses de l'impact des interférences. Une autre manière d'aborder le problème est d'empêcher les interférences en améliorant l'isolation entre les différents cœurs. Loin de concurrencer les analyses susmentionnées, l'isolation peut offrir des hypothèses plus favorables à l'estimation de l'impact des interférences. Nous reprenons la distinction utilisée pour catégoriser les interférences, en distinguant l'*isolation spatiale* de l'*isolation temporelle*.

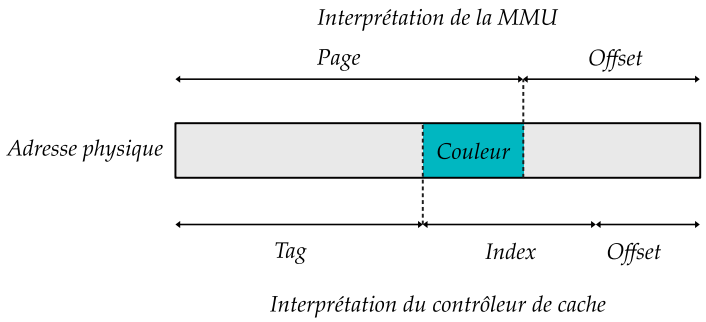


FIGURE 3.5 – Bits de couleurs utilisés pour la coloration de cache

Isolation spatiale

L'isolation spatiale est utilisée pour partitionner les différentes mémoires entre les cœurs. On peut mettre en œuvre cette isolation par des moyens logiciels grâce à la coloration de pages, mais aussi à l'aide de support matériel, notamment dans les caches [67].

Coloration de pages La coloration de page [94] désigne les politiques d'allocation de pages physiques reposant sur la notion de *couleur*. Un allocateur de page utilisant la coloration de page, associe à une adresse de page virtuelle une adresse de page physique dotée d'une couleur encodée à l'aide de certains bits d'adresse physique. Cette technique a été utilisée en mono-cœur pour optimiser le placement de données dans les caches indexés physiquement [50, 84, 90, 29]. En définissant la couleur d'une page à l'aide des bits à la fois utilisés pour définir l'adresse de page et ceux utilisés pour l'indexation dans le cache (Figure 3.5), il est possible de contrôler le placement des données dans celui-ci. En particulier, on a la garantie que deux pages avec des couleurs distinctes ne peuvent occuper les mêmes sets dans le cache. En allouant des pages de couleurs différentes aux pages virtuelles adjacentes, on peut donc s'assurer que les données sont réparties uniformément dans le cache, limitant ainsi les défauts au sein d'une même application. En multi-cœur, la coloration de page permet de partitionner des ressources en attribuant des ensembles disjoints de couleurs pouvant être alloués aux différents cœurs. Cette approche a été utilisée pour réduire le problème de la pollution dans les caches partagés [91]. La coloration de page peut être appliquée à d'autres ressources matérielles, notamment les bancs mémoires [107], les canaux DDR [70] ou encore les entrées de TLB [75].

En pratique, l'application de la coloration de page se heurte à plusieurs difficultés.

Tout d'abord, pour utiliser cette technique il faut savoir comment les adresses sont interprétées par les différents composants que l'on souhaite partitionner. Cette information n'est pas toujours disponible. Des approches de rétro-ingénierie [107, 75] ont néanmoins été proposées pour pallier ce manque d'information. Elles se basent sur l'étude du temps de réponse d'accès mémoire en fonction des adresses accédées. L'arrivée des fonctions de hachages pour indexer les caches pose de nouveaux problèmes à la mise en œuvre de la coloration de cache. Ces fonctions transparentes pour l'utilisateur ont une influence non négligeable sur les performances. Ainsi, elles constituent souvent un secret industriel et doivent être identifiées par rétro-ingénierie [106]. De plus, même en connaissant la fonction de hachage, il n'est à connaissance pas possible de colorer des caches indexés de la sorte.

La coloration de pages est un mécanisme nécessitant du support au niveau de l'allocateur de pages physiques du système d'exploitation. Or, ce mécanisme a des inconvénients pouvant dissuader les développeurs de systèmes d'exploitation d'implanter le support nécessaire à sa mise en œuvre. C'est par exemple le cas dans le noyau LINUX [95]. Les raisons invoquées sont une allocation de pages physiques rendue plus complexe et de possibles augmentations de la pression mémoire [4]. Pour employer la coloration de pages sur ces systèmes, il faut donc utiliser un allocateur de pages spécifique. À cette fin, Yun et al. ont proposé PALLOC [107], un allocateur de page pour LINUX gérant la coloration de pages.

Support matériel Certaines architectures offrent du support matériel pour l'isolation spatiale, notamment dans les caches avec les fonctionnalités de verrouillage de lignes et de verrouillage de voies. Le verrouillage de lignes permet de protéger spécifiquement une donnée. Tandis que le verrouillage par voie permet d'empêcher la sélection de certaines voies par l'algorithme de remplacement. Cette dernière méthode est généralement préférée, dans la mesure où elle ne nécessite pas de spécifier quelles données sont verrouillées. Le verrouillage de voies peut être utilisé conjointement à la coloration de page afin de permettre un partitionnement à la granularité d'une page [66].

Conclusions

Dans les systèmes temps-réel, le gain de déterminisme apporté par l'isolation spatiale a des avantages, en permettant par exemple de simplifier certaines analyses, mais aussi

4. Zhang et al. ont montré que ce dernier point pouvait être évité en n'employant la coloration que sur les pages les plus utilisées [110]. Bien que prometteuse, cette solution rend d'autant plus complexe l'allocation de page physique.

des inconvénients. L'isolation spatiale apporte un gain de déterminisme qui peut être coûteux en termes de performance en isolation. Ce compromis a été étudié par Kim et al. [53] et Blin et al. [24], qui ont tous les deux conclu à de meilleures performances avec l'isolation spatiale. Un autre problème est la gestion des couleurs au sein du système. À cet effet, Ward et al. propose de considérer les couleurs comme des ressources partagées. Et propose une analyse d'ordonnancement associée, se basant sur le principe d'équivalence avec un système mono-cœur [99].

Isolation temporelle

L'isolation temporelle a pour but d'empêcher l'accès simultané à une ressource, en allouant des fenêtres de temps à chaque cœur (polititque TDMA [5]). Nous ne parlerons ici que des méthodes logicielles, que l'on peut qualifier de méthodes de contrôle [36].

Une première approche de contrôle est l'approche tout ou rien, illustrée dans la figure 3.6. Celle-ci s'applique dans les systèmes à criticité mixte [97], dans lesquels des tâches non critiques peuvent être soumises à des interférences (les tâches $PR1, PR3, PR4$ de la figure 3.6) et des tâches critiques doivent absolument être isolées (la tâche $PR2$). Cette approche peut être décrite par la règle suivante :

1. Si une tâche critique est ordonnancée sur un cœur, les tâches de tous les autres cœurs doivent être suspendues.
2. Lorsque des tâches non critiques s'exécutent, les tâches critiques sont suspendues.

Ainsi, on peut s'assurer que les tâches critiques ne souffrent pas d'interférences. Cette approche a notamment été utilisée dans l'hyperviseur temps-réel PIKEOS [5] pour obtenir , sur un processeur bi-cœur, le plus haut niveau de certification dans un contexte ferroviaire [34].

Si elle offre une isolation parfaite aux applications critiques, la solution employée dans PIKEOS ne permet pas aux applications critiques de bénéficier du parallélisme. Une meilleure utilisation des ressources est possible en faisant des hypothèses sur l'exécution des tâches. Des modèles d'exécution, comme le superblock [31, 85], ou PREM [76] permettent de diviser les applications en phases de communication et en phases d'exécutions. Durant les phases de communications, l'application peut utiliser des ressources partagées. Tandis que dans les phases d'exécution, l'application n'utilise que des ressources locales à un cœur. Un ordonnancement des différentes phases est ensuite effectué, en permettant l'exécution en parallèle des phases d'exécutions et en

5. Time Division Multiple Access

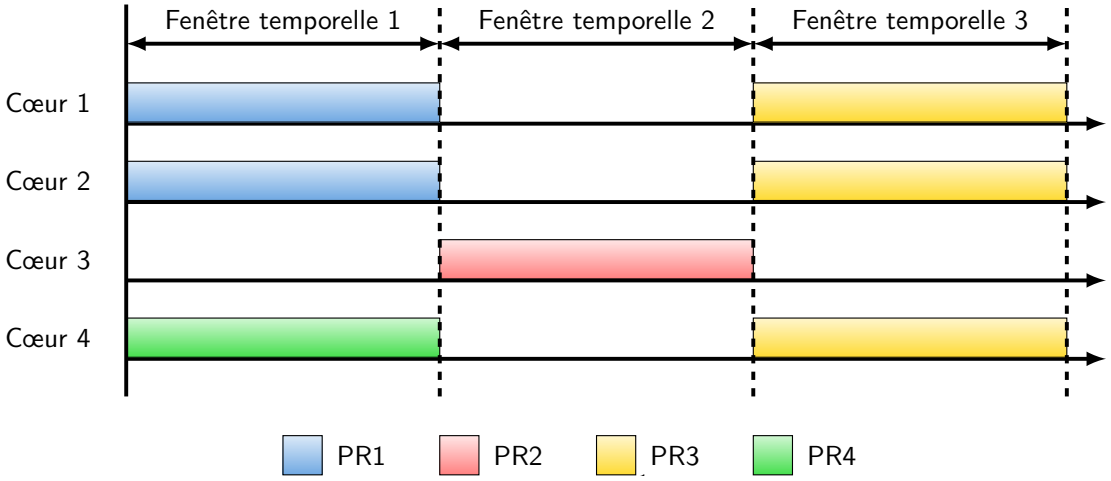


FIGURE 3.6 – Ordonnancement tout ou rien

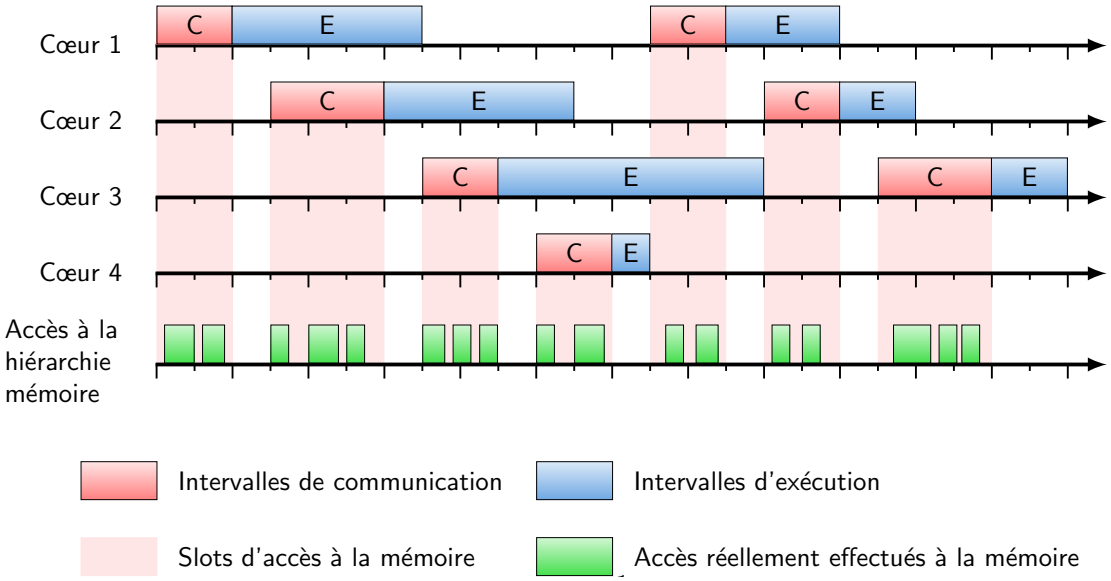


FIGURE 3.7 – Ordonnancement d'une tâche avec un modèle d'exécution déterministe

s'assurant de l'exécution séquentielle des phases de communications (figure 3.7). Ce type de modèle a été utilisé dans de multiples travaux sur l'ordonnancement dit *memory centric* [105, 18, 11]. Rivas et al. [83] détaillent l'implantation du support pour le modèle PREM, dans le système d'exploitation temps-réel multi-cœur Hiperros [3] [83]. Ce type de modèle permet d'améliorer la granularité de l'isolation temporelle, en permettant le parallélisme sur les phases d'exécutions. Il nécessite par contre des adaptations du logiciel, rendant ce type d'approche incompatible avec des logiciels patrimoniaux.

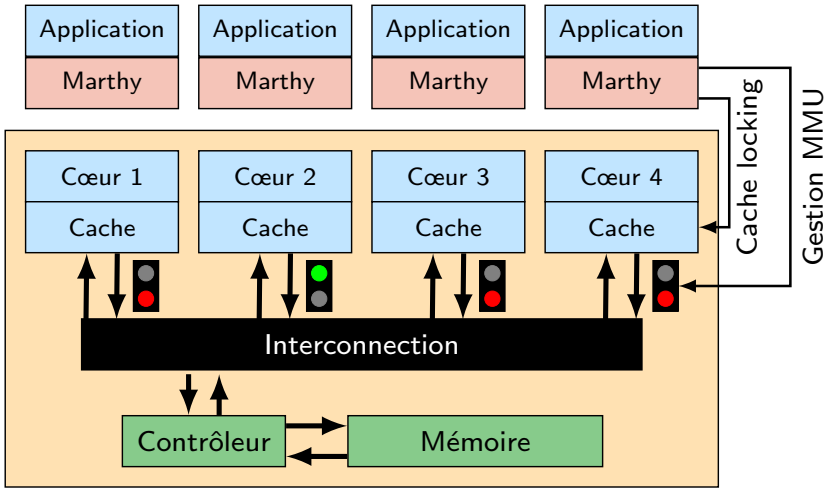


FIGURE 3.8 – MARTHY

MARTHY [48] est un logiciel de contrôle permettant de mettre en œuvre une politique d'accès TDMA vers la mémoire dans un hyperviseur. Le contrôle des accès est réalisé à l'aide d'une stratégie de programmation de la MMU et de fonctionnalités de verrouillage de cache. L'idée est de maintenir l'invariant que toute page disposant d'une traduction valide est présente et verrouillée dans le cache. Lorsqu'une application tente d'accéder à une donnée qui n'est pas présente en cache, une interruption de défaut de page est levée :

1. Une exception de défaut de page est levée, donnant ainsi la main à l'hyperviseur (Marthy).

2. Un emplacement pour accueillir la page à charger est sélectionné. Dans la majeure partie des cas, cette étape consiste à sélectionner une page à évincer.

3. L'hyperviseur attend le début d'une fenêtre temporelle durant laquelle il est autorisé à accéder à la mémoire.

4. La page est chargée durant la fenêtre.

MARTHY permet donc de bénéficier des avantages apportés par les modèles d'exécutions déterministes sur tout type d'application. Cette approche néanmoins dégrade significativement les performances lorsque le mécanisme de contrôle est très sollicité. De plus, sa mise en œuvre nécessite des fonctionnalités de verrouillage de cache, qui ne sont pas toujours disponibles.

Conclusions

L'isolation temporelle a l'avantage d'offrir beaucoup de déterminisme en éliminant les accès simultanés aux ressources partagées. Ce déterminisme est néanmoins acquis au prix d'une sous utilisation du matériel. Elle s'avère néanmoins difficile à mettre en œuvre. Les modèles d'exécutions déterministes sont une piste prometteuse pour les applications futures. Néanmoins, il n'est pas certain que toutes les applications puissent être écrites pour satisfaire à ce genre de modèle. Une question demeure sur la granularité du contrôle des accès. Si une granularité plus fine peut permettre une meilleure utilisation du matériel, le coût du contrôle augmente également. À notre connaissance, il n'y a pour le moment pas de réponse claire au problème du coût de contrôle dans la littérature.

3.4.4 Systèmes de régulation

L'isolation des différents cœurs, qu'elle soit spatiale ou temporelle, offre du déterminisme au prix d'une dégradation des performances. Selon le niveau de charge du système, le gain de déterminisme peut ne pas justifier ce coût. Le but des *systèmes de régulation* est d'adapter le niveau d'isolation pour les tâches temps-réel en fonction de la pression effectivement exercée sur les ressources partagées. Nous distinguons deux composants majeurs dans ces systèmes :

- Un *plan de données*, en charge de déterminer s' il y a un risque de dépassement d'échéance en fonction de l'état observé du système.
- Un *plan de contrôle*, en charge de rétablir un domaine de fonctionnement sûr en cas d'alerte du plan de données.

Ce type de solutions est plutôt adapté à des systèmes à criticité mixte, car dans ce type de systèmes des tâches non critiques peuvent être suspendues pour permettre aux taches critiques de respecter leurs échéances.

Une première approche consiste à réguler l'accès aux ressources. C'est ce que permettent les *régulateurs de bandes passantes*, tels que MRS [46] ou MemGuard [108]. Ces régulateurs fonctionnent sur le principe d'un *serveur périodique*. Chaque cœur se voit attribuer un budget d'accès autorisés sur une *période de régulation*. Les éventuels dépassements de budgets sont détectés à l'aide de *compteurs de performances*. Des compteurs de performances sont utilisés pour détecter le dépassement de budget éventuel. En pratique, ces compteurs sont programmés pour lever une interruption lorsqu'ils sont saturés et

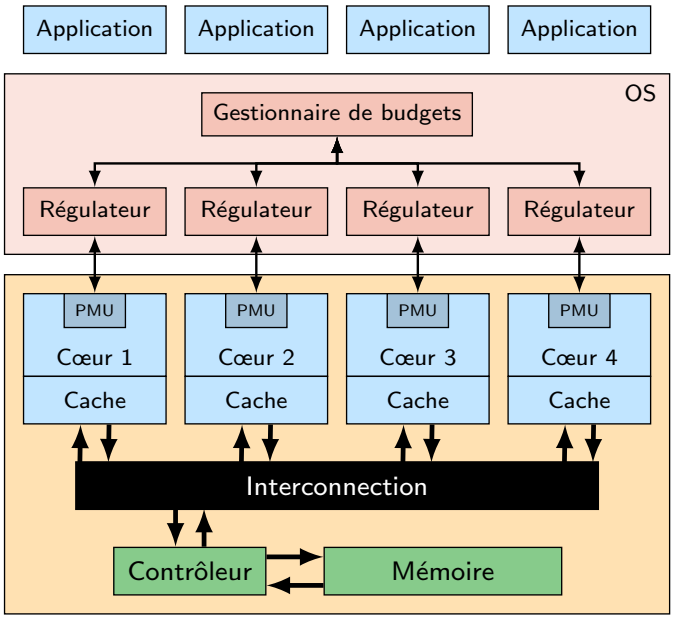


FIGURE 3.9 – MemGuard

initialisés à leur valeur maximale moins le budget alloué. Lors d'un dépassement de budget, le cœur concerné est mis en attente jusqu'à la période de régulation suivante.

Ce type d'approche peut être utilisée pour simplifier l'analyse d'ordonnancement d'un système [77, 9]. La mise en œuvre d'un régulateur de bande passante dépend directement des événements observables sur le matériel utilisé. Notons que cette observation doit se faire indépendamment pour chaque cœur. Ce n'est pas toujours possible sur les cibles COTS. En effet, dans ces dernières, les ressources partagées ne sont généralement observables qu'au moyen d'un compteur global ne permettant pas de différencier la consommation des différents cœurs.

De plus, le bon fonctionnement de ce type d'approche dépend directement du dimensionnement du système et donc des budgets alloués aux différents cœurs. En particulier, il n'y a pas de notion de sensibilité. C'est-à-dire que si le dimensionnement du système est trop optimiste, rien ne garantit le respect des échéances temporelles. MemGuard apporte une solution à ce problème en définissant un seuil maximal d'utilisation du bus mémoire permettant de garantir le respect des échéances, ce qui pose un problème de sous utilisation des ressources.

Le *Contrôleur de WCET distribué* [56, 57] est une approche de régulation dont le plan de contrôle n'est pas focalisé sur l'utilisation des ressources, mais sur le retard pris par

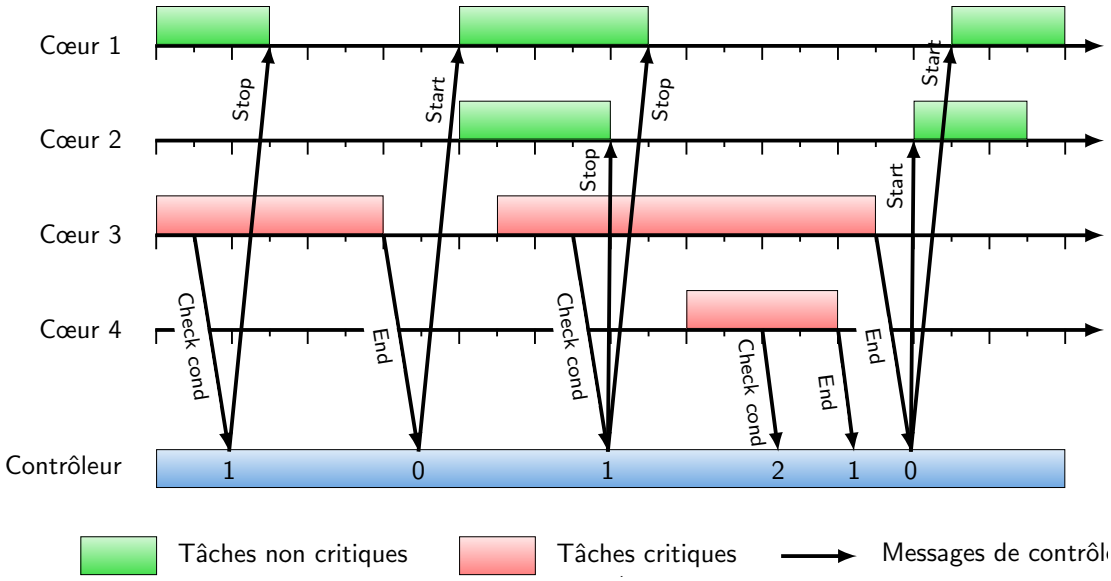


FIGURE 3.10 – Contrôleur de WCET distribué

les tâches temps-réel. Cette approche concerne les systèmes à criticité mixte. Des *points de contrôles* sont injectés dans les applications temps-réel du système. Ces points de contrôle comprennent une *condition de sûreté* permettant de vérifier que la tâche peut respecter ses échéances malgré la contention. Si la condition de sûreté d'un point de contrôle n'est pas respectée, alors une notification est envoyée à un *contrôleur de WCET* qui en réponse va suspendre les tâches non critiques du système. La principale difficulté dans la mise en œuvre de ce type d'approche est le placement des points de contrôles et la spécification des conditions de sûreté.

La solution proposée par Blin et al. [24] vise à maintenir le retard subi par une application en dessous d'un seuil déterminé lors de la conception du système. Tout comme le contrôleur de WCET distribué, cette approche s'applique à des systèmes à criticité mixte et suspend les tâches non critiques lorsque le seuil de retard spécifié est atteint. Ce mécanisme n'exige pas compteurs locaux, mais un compteur global, la rendant compatible avec la plupart des cibles COTS. Elle est néanmoins limitée au cas où seul un cœur exécute des applications temps-réels. Ce système repose sur trois grandes étapes. Deux étapes hors-ligne permettent de définir le plan de données, tandis que la troisième étape en ligne implante le plan de contrôle :

1. *Modèle de retards de la plateforme matérielle*. Cette étape consiste à définir une fonction, permettant d'associer un surcoût temporel à une consommation de bande passante en isolation et une bande passante globale observée. Cette fonction est conservative

et construite à partir de données expérimentales. De multiples expériences sont effectuées avec des microbenchmarks, et seuls les cas les plus pessimistes sont retenus.

2. *Modèle de consommation de l'application à protéger*. Dans cette étape, la consommation de bande passante d'une application est mesurée en isolation. Les mesures sont effectuées sur différentes phases de l'application.

3. *Contrôle en ligne* À l'exécution, un mécanisme de contrôle mesure la bande globale et détermine le retard subi par l'application temps-réel en utilisant les deux modèles constituant le plan de données. Lorsque le retard cumulé est trop élevé, les applications non temps-réels sont suspendues.

La mise en œuvre efficace de ce type d'approche dépend grandement de la qualité du plan de données que l'on a pu définir. Plus le modèle de retards est imprécis, plus vite les applications non critiques sont suspendues, ce qui résulte dans une utilisation moindre du matériel. Une meilleure précision du modèle de retards peut donc conduire à une meilleure utilisation du matériel.

Conclusions

Les systèmes de régulation permettent d'adapter le niveau d'isolation du système en fonction de l'utilisation globale des ressources matérielles. Ces approches offrent donc un compromis entre déterminisme et efficacité, ce qui les rend particulièrement attractives pour des applications industrielles. L'implantation efficace de tel système demeure néanmoins un défi important. Un verrou technologique important réside notamment dans le plan de données en charge de détecter les situations à risque. En effet, pouvoir détecter précisément ces situations détermine directement la confiance que l'on peut avoir dans ces approches et éviter les surdimensionnements excessifs.

3.5 Conclusions du chapitre

Nous avons dans ce chapitre présenté l'impact des interférences dans le cadre des systèmes temps-réels. Prendre en compte les interférences lors de la conception des systèmes temps-réels revient à surdimensionner ces derniers. En effet, le dimensionnement de ces systèmes doit correspondre aux pires cas. La complexité et l'opacité du matériel moderne rendent ces interférences difficilement prévisibles et conduisent à des surdimensionnements inacceptablement pessimistes des systèmes.

À l'heure actuelle, les approches traditionnellement employées pour l'analyse de pire temps d'exécutions sont mises en échec par le problème des interférences. Il est possible dans certains cas d'améliorer l'isolation des différents, mais cela se fait au détriment des performances. Enfin, des approches de régulation permettent d'obtenir un compromis intéressant entre déterminisme et performances, mais leur mise en œuvre nécessite d'anticiper efficacement le retard subi à cause des interférences.

Dans le reste de ce document, nous présentons nos travaux dont le but est de caractériser la sensibilité aux interférences des applications à partir d'une caractérisation de leur comportement exécutée seules. Le but est de pouvoir estimer à priori le retard que peut subir un programme, mais aussi de permettre, à terme, la conception de systèmes de régulation efficaces.

Deuxième partie

Contributions

CHAPITRE 4

Évaluer l'impact des interférences mémoires

Dans ce chapitre, nous nous penchons sur l'étude empirique de l'ampleur du phénomène d'interférences sur une carte multi-cœur COTS. Cette étude a non seulement pour but de déterminer l'impact des ralentissements subis, mais également à quel point ces ralentissements peuvent varier. Cela pose deux difficultés auxquelles nous allons apporter des solutions dans ce chapitre. La première étant d'identifier les aspects pertinents du trafic mémoire pour le problème d'interférences. La seconde est d'avoir un ensemble d'applications représentatives de ces différents aspects.

Ce chapitre est organisé en trois sections. Dans la première section, nous présenterons la plateforme matérielle de référence sur laquelle nous conduirons le reste de nos travaux. Dans la deuxième section, nous présenterons un modèle événementiel nous permettant d'identifier différents aspects du comportement d'accès à la mémoire d'un programme. Nous introduirons, dans la section suivante, un ensemble de microbenchmarks paramétrables permettant de générer différents types de trafic. Enfin, dans la quatrième section, nous utiliserons ces microbenchmarks pour étudier l'impact des interférences sur notre matériel de référence.

4.1 Plateforme matérielle

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

Nous utilisons une carte i.MX6Q Sabre Lite comme plateforme matérielle de référence pour nos travaux. Ce choix est motivé par plusieurs raisons. Tout d'abord, nos travaux s'inscrivant dans le cadre d'une thèse CIFRE, le matériel choisi se devait d'être représentatif de celui utilisé dans l'industrie. C'est le cas de cette carte, initialement conçue pour des applications multimédias. Une variante automobile [88] de cette carte a d'ailleurs été largement utilisée par un grand nombre de fabricants et de fournisseurs de l'industrie automobile. La deuxième raison qui motive notre choix est que cette carte offre un bon niveau de documentation.

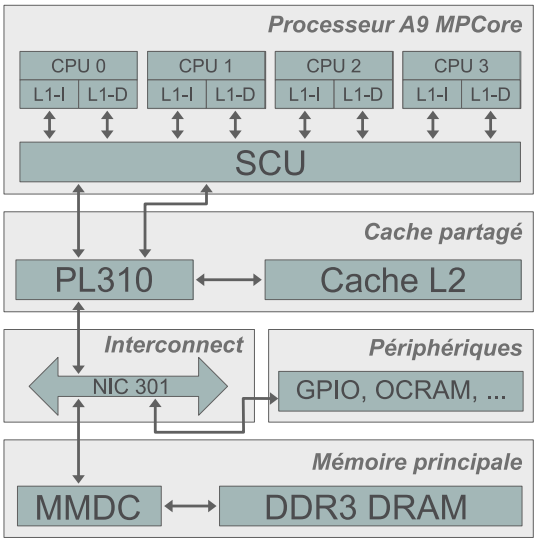


FIGURE 4.1 – Architecture matérielle simplifiée de la carte SABRE Lite

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 mémoire *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 amorce 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 plateforme, 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*.

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 suivant l'architecture *ARMv7-A* [13] et les jeux d'instructions 32-bit ARM, 16-bit et 32-bit Thumb [16].

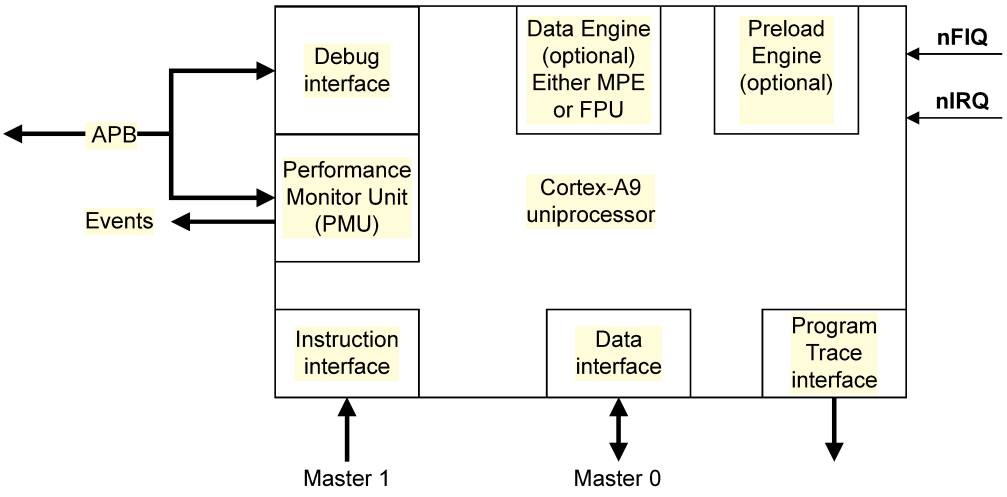


FIGURE 4.2 – Système monoprocesseur Cortex-A9 [16]

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. Nous utiliserons ces compteurs pour caractériser en partie le comportement des applications.

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*.

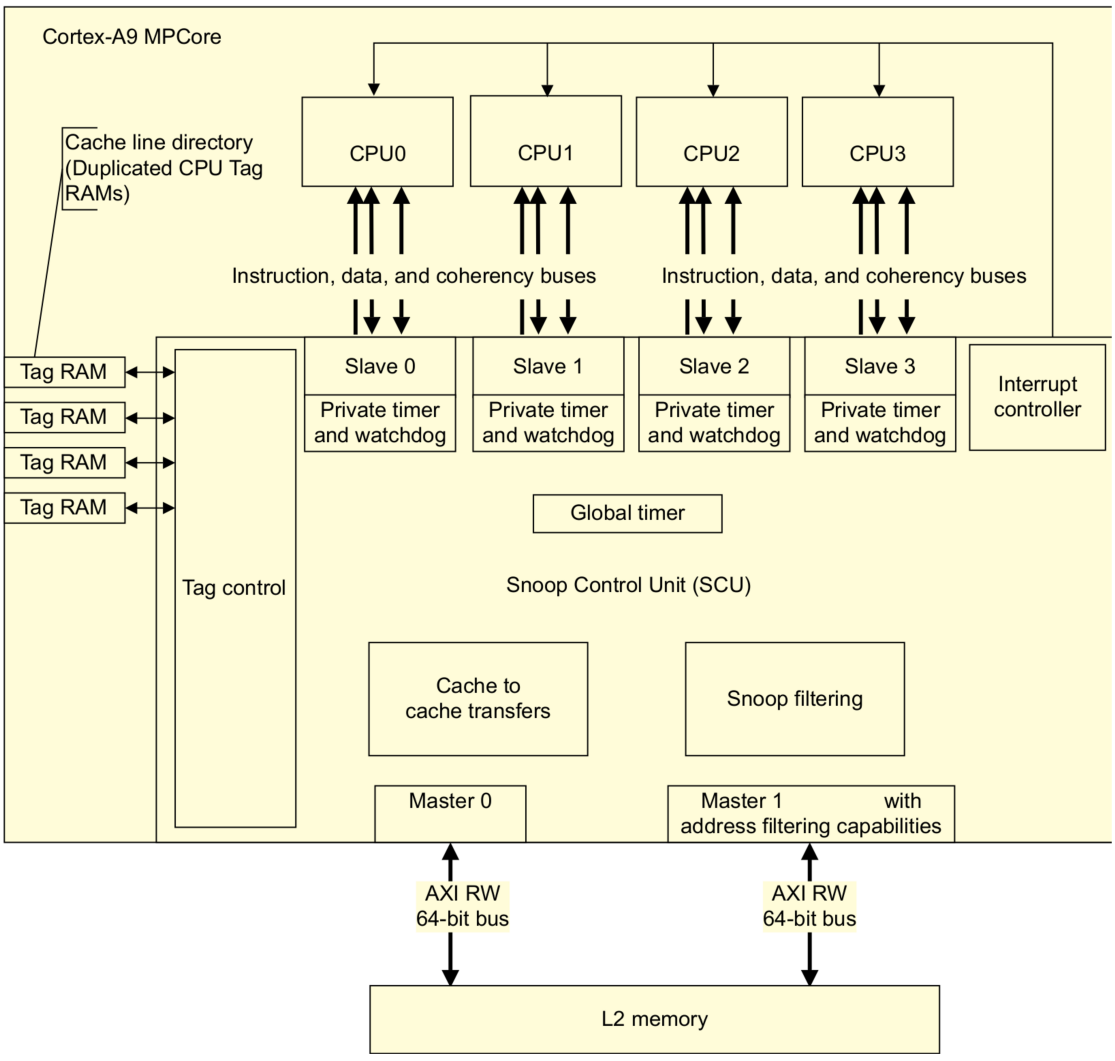


FIGURE 4.3 – Exemple de configuration multiprocesseur [15]

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 un ensemble de périphériques mappés en mémoire incluant un temporisateur global, ainsi qu'un *watchdog* et un temporisateur privé pour chacun des processeurs du groupe. Un contrôleur d'interruptions respectant l'architecture *Generic Interrupt Controller* [17] 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*).

Impact sur les interférences

Bien que les cœurs Cortex-A9 sont indépendants les uns des autres, un couplage existe au travers de l'interface MPCore, et plus particulièrement la *SCU*. Celle-ci a deux fonctions : maintenir la cohérence des données entre les caches L1 des différents cœurs et arbitrer l'accès au cache L2.

Le maintien de la cohérence entre les cœurs entraîne une augmentation du nombre d'invalidations lorsque des données sont partagées. Il s'agit d'une forme d'interférence spatiale qui est propre aux applications utilisant le modèle SMP. Vu que nos travaux se concentrent sur le modèle AMP, nous ne tiendrons pas compte de ce type d'interférence.

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 répondre en parallèle aux requêtes d'accès étant limitée, l'ordre d'émission des requêtes vers la hiérarchie mémoire de niveau supérieur fait l'objet d'une politique d'arbitrage. Cette politique n'est pas détaillée dans la documentation matérielle dont nous disposons. Il s'agit d'un problème récurrent dans les architectures COTS.

Les sections suivantes s'attachent à décrire plus en détail les caches de premier niveau qui sont intégrées 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 32KiB [87, 16]. Un cache est utilisé pour contenir les instructions de code, l'autre pour charger les données. La politique de correspondance utilisée dans ces deux caches est de type « partiellement associative ». Ainsi, chaque cache est 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 [14].

Cache d'instruction

Politique d'indexation 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. Le code des programmes exécutés étant majoritairement en lecture seule, l'utilisation d'une telle politique sur le cache d'instruction n'est donc pas rédhibitoire.

Politique de remplacement La politique de remplacement du cache peut être, au choix, pseudo *round-robin* ou *pseudo-random*.

Optimisations Le cache L1 est connecté à une unité désactivable de prédiction du flux d'instructions du programme qui est utilisé pour précharger en avance les instructions qui vont être exécutées.

Cache de données

Politique d'indexation 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.

Politique de remplacement La politique de remplacement des données utilisées par le cache est fixée en dur à *pseudo-random*.

Optimisations 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 *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 soit bloqué le temps de la propagation.

Impacts sur les interférences

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. Si l'ensemble de l'empreinte mémoire d'une application est suffisamment faible pour être contenue dans le cache, alors le problème de contention ne se pose plus.

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 a pour effet d'entraîner un accroissement ponctuel de la demande de bande passante mémoire se traduisant par une augmentation possible des interférences. Ces mécanismes ont également pour effet de rendre plus difficile l'estimation du trafic effectivement généré vers les niveaux supérieurs de la hiérarchie mémoire.

4.1.3 Cache L2

Le cache de niveau 2, d'une taille d'un mégaoctet [87], 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 [86]. 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 [14].

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 maîtres, permettant de servir quatre caches MISS en parallèle, de trois *eviction buffer*, de la taille d'une ligne de cache, utilisé pour stocker les lignes évincées en attente de leur propagation vers la DRAM et de trois *store buffer* de 32 bytes capables 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, transférées 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.

4.1.4 Contrôleur de cache L2

Le contrôleur de cache PL310 est doté d'un mécanisme 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 gérées 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 politique 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 leurs 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 MMDC par deux bus AXI 64 bits connecté au bus d'interconnexions NIC-301.

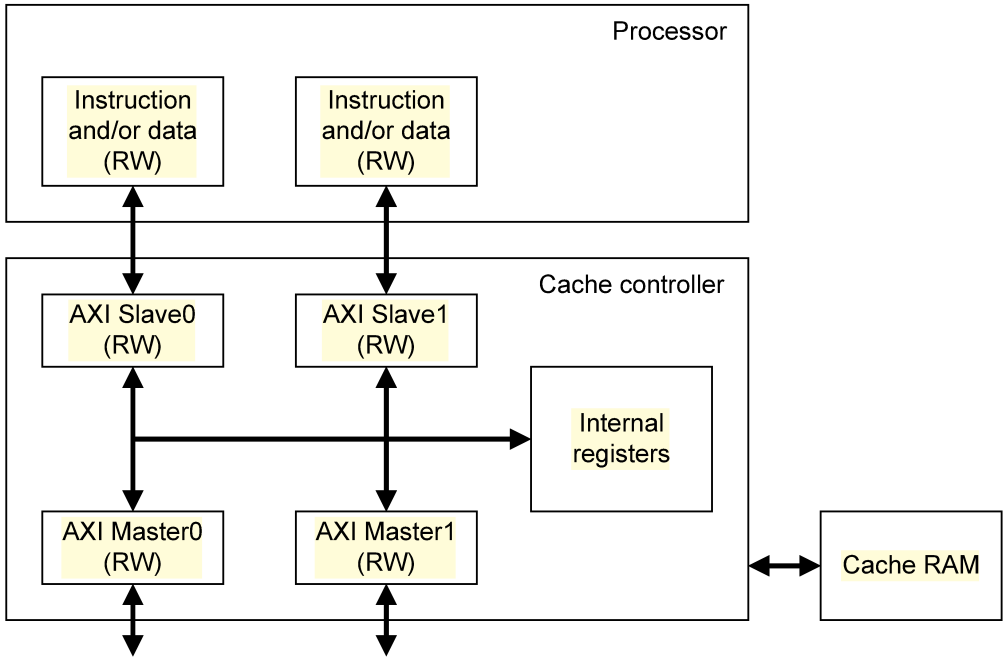


FIGURE 4.4 – Exemple de contrôleur de cache interfacé avec un processeur ARM [86]

Impact sur les interférences

Le cache de deuxième niveau étant partagé entre les quatre cœurs du processeur, il est un canal pour deux types d'interférences.

Tout d'abord, les caches partagés sont un canal d'interférence spatiale. Une application ordonnancée sur un cœur peut en effet évincer à son profit les données d'un applicatif ordonnancé sur un autre cœur. Il s'agit d'une des interférences le plus souvent mises en avant dans la littérature. Elles peuvent néanmoins être prévenues avec de la coloration de pages ou de manière équivalente avec la fonctionnalité de *lockdown by master* fournie par notre carte.

Le deuxième type d'interférences est temporel et à lieu au niveau du contrôleur PL310. En effet, ce dernier peut traiter un certain nombre de requête en parallèle. Lorsque sa capacité de traitement est dépassée, les requêtes sont traitées séquentiellement. Une politique d'arbitrage doit alors être appliquée sur les requêtes à traiter. Or, cette dernière n'est malheureusement pas documentée.

4.1.5 Contrôleur mémoire

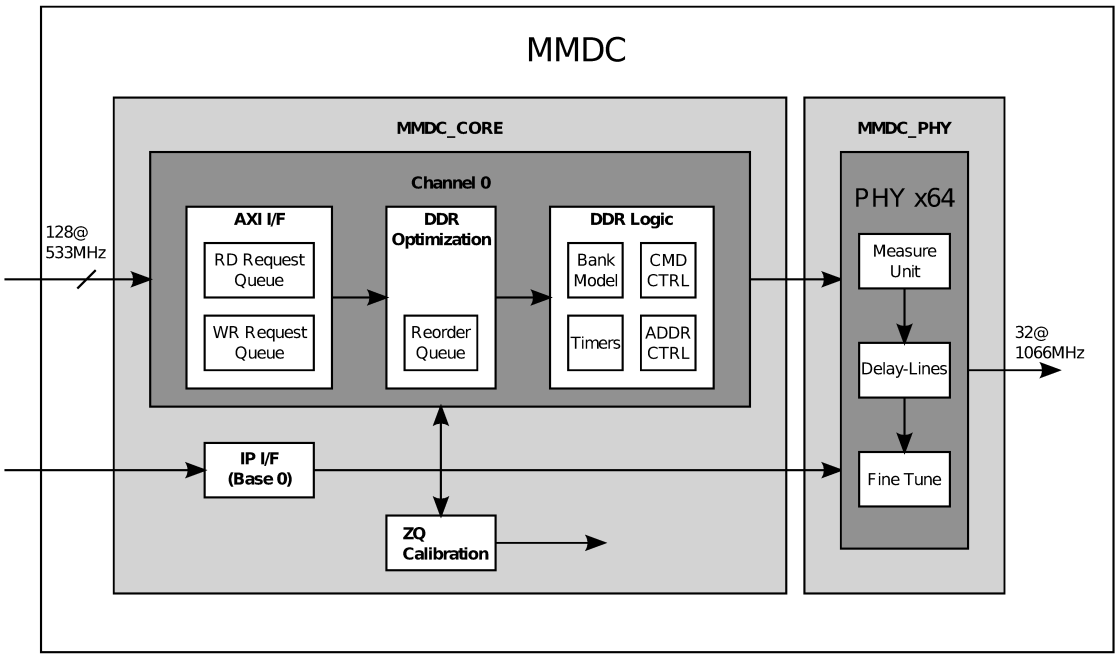


FIGURE 4.5 – Diagramme de bloc du contrôleur MMDC [87]

Le contrôleur mémoire MMDC (*Multi Mode DDR Controller*) conçu par Freescale est chargé de gérer la mémoire DRAM de la plateforme matérielle. Il est constitué de deux composants, le *cœur*, 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 [12] 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 réordonnancement.

Un mécanisme d'arbitrage est utilisé pour élire une requête au sein du tampon de réordonnancement 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 réordonnancement. Le contrôleur mémoire met en œuvre une politique de gestion des *row-buffer* de type *open-row* que nous avons précédemment décrite en section 4.1.5. Le décodage d'adresses utilise une politique de *bank interleaving* dans laquelle les lignes consécutives en mémoire sont placées dans des bancs consécutifs.

Afin d'optimiser l'utilisation du bus DDR, les requêtes d'accès sont réordonnancées dans le tampon de réordonnancement. Les requêtes présentes dans le tampon se voient attribuer un score afin de déterminer leur priorité, la requête avec le score le plus élevé est sélectionnée pour être envoyée vers le composant *DDR logic*. Le score final est calculé en additionnant quatre facteurs différents et en tronquant les quatre bits de poids faibles :

1. *Dynamic jump score* Ce score est incrémenté à chaque fois qu'une requête n'est pas sélectionnée. Ce score a une valeur maximale, égale à 15 sur notre cible. Lorsque cette valeur est atteinte, un mécanisme anti-famine est activé.
2. *Page hit score* Il s'agit d'un score statique attribué lorsque la ligne de destination de la requête est chargée dans un row buffer. Sur notre cible, ce score est égal à 4.
3. *Access score* Il s'agit d'un score statique attribué lorsque le type de l'accès (lecture ou écriture) est le même que celui de la requête sélectionnée précédemment. Sur notre cible, ce score est égal 2.
4. *QoS score* Il s'agit ici d'un score attribué dynamiquement par le matériel. Il est encodé sur 4 bits et ne peut donc pas dépasser 15. Mis à part cela, son mode de calcul n'est pas connu. Lorsque le mode temps réel du contrôleur mémoire est activé, toutes les requêtes avec un score QoS maximal deviennent prioritaires sur les autres requêtes.

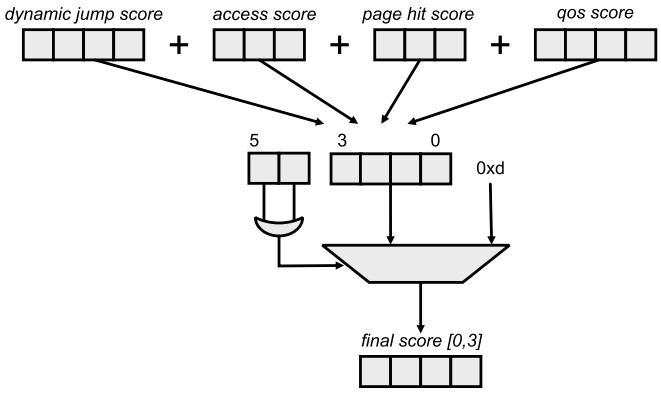


FIGURE 4.6 – Calcul de la priorité lors du réordonnancement des accès DRAM

Lorsque le *dynamic jump score* d'un accès atteint la valeur maximale, un mécanisme anti-famine est activé : un second compteur est incrémenté, jusqu'à atteindre une valeur maximale (fixée à 15 sur notre plateforme). Lorsque le compteur associé à une requête atteint une valeur maximale, cette requête prend la priorité sur toutes les autres.

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 utilisée afin de préparer en avance la gestion des futurs accès. Selon la documentation [45], la prédiction se fait en utilisant notamment en considérant les accès au niveau du processeur, et du tampon de réordonnancement.

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, ...).

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 mémoires peut donc remplir les tampons du contrôleur retardant ainsi le traitement des requêtes des autres demandeurs.

Une autre source d'interférences réside dans le réordonnancement des requêtes d'accès. En effet, le contrôleur cherche à maximiser les performances globales de la machine tout en évitant les conflits entre requêtes. Pour y arriver, le contrôleur mémoire réordonnance les requêtes de telle sorte à maximiser le débit mémoire total de la plateforme 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 une combinaison 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.

Enfin, 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.

4.1.6 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.

Composant		**Taille**	**Correspondance**	**Remplacement**	**Métriques Disponibles**
Cache L1	Donnée	32KiB	4	PLRU ou PR	PMU locale
	Instructions		4	PR	
Cache L2		1MiB	16	PLRU ou PR	PMU globale
			Nombre de bancs		
Mémoire principale		1GiB	8	N/A	PMU globale

TABLE 4.1 – Récapitulatif des caractéristiques matérielles des différents niveaux de la hiérarchie mémoire.

4.2 Un modèle événementiel du trafic mémoire

Pour étudier la sensibilité des programmes aux interférences mémoires, nous devons représenter l'interaction entre une application et le système mémoire. Nous adopterons à cette fin un *modèle événementiel* du trafic mémoire représentant le *flux explicite de requête d'accès vers un système mémoire partagé* qui sont générées lors de l'exécution d'un programme.

L'exécution d'un programme est représentée par sa *trace*, c'est à dire par la suite d'instructions qui ont été exécutées. Une trace correspond au parcours d'un chemin dans le graphe de flots de contrôle. Nous supposerons toujours la terminaison du programme, et donc que le nombre d'instructions dans une trace est fini. Nous noterons celui-ci N_{inst}.

L'exécution d'une instruction entraîne une interaction avec le matériel, qui peut impliquer des éléments partagés du système mémoire. Afin de ne pas modéliser le matériel en détail, et dans un souci de généricité, ces éléments sont regroupés dans une *boite noire* représentant la mémoire partagée dans son ensemble. Lorsqu' une instruction interagit avec un composant appartenant à cette boite noire, une *requête d'accès* est émise vers celle-ci. Une requête d'accès peut être émise dans deux cas précis :

- Lors d'une instruction d'accès à la mémoire.
- Lors du chargement d'une instruction.

Nous pouvons ainsi associer à toute trace d'exécution une suite de requêtes d'accès émises. Cette suite est également finie, et nous notons son nombre d'éléments N_{access}. Une requête mémoire est caractérisée par un *sens*, une instruction de source et une adresse de destination.

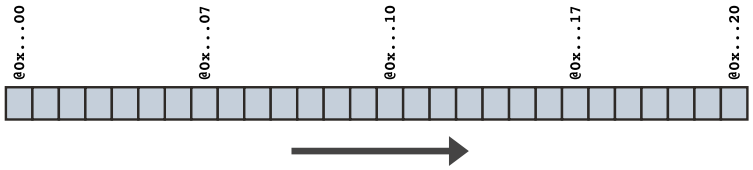
Ce modèle représente le trafic généré *explicitement* par l'application. Sur un processeur moderne, le trafic généré peut être différent. Il y a deux raisons à cela. D'une part, les instructions peuvent être exécutées dans le désordre. D'autre part, les mécanismes spéculatifs (prédiction de branchements, préchargements de données) peuvent également entraîner des accès qui ne sont pas représentés par le modèle. Ce trafic *implicite* varie fortement en fonction du matériel considéré. Pour le prendre en compte, il faudrait pouvoir modéliser finement ce dernier, ce qui serait incompatible avec une vision boite noire de celui-ci.

La trace considérée correspond à la suite d'instruction obtenue en parcourant le graphe de flot de contrôle du programme. Sur un processeur moderne, ces instructions peuvent être exécutées dans un ordre différent. De plus, les mécanismes de spéculation

(prédiction de branchements, préchargement de données) peuvent causer des accès supplémentaires qui ne sont pas capturés. C'est une limitation de cette représentation.

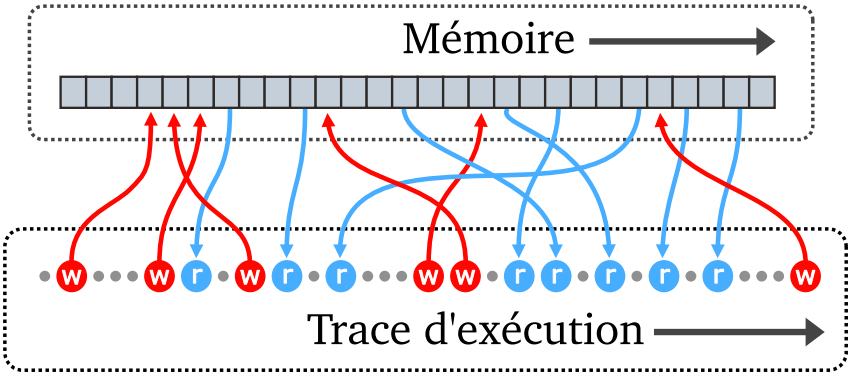
Nous aurons par la suite souvent recours à une représentation graphique de l'activité mémoire. La trace d'exécution y est représentée par une suite de points colorés. Les instructions ne déclenchant pas d'accès vers la mémoire y sont représentées par des petits points gris, tandis que les déclenchant des accès sont représentés par de gros points colorés (bleu pour les requêtes en lecture et rouge pour les requêtes en écriture). La flèche indique le sens d'exécution [1]. Cette représentation nous donne de premières indications sur les variations de comportements d'accès à la mémoire. Cette représentation de la trace permet d'identifier aisément plusieurs caractéristiques du comportement d'accès à la mémoire. La proportion d'accès à la mémoire par rapport au nombre total d'instructions exécutées indique l'intensité de l'utilisation de la mémoire. La représentation des types d'accès permet de caractériser la proportion de lectures et d'écritures, mais aussi de leur entrelacement.

Nous pouvons également représenter l'interaction avec la mémoire. Cette dernière est représentée par un tableau de case mémoire. Chaque case est identifiée par une adresse. Comme pour la trace d'exécution une flèche indique dans quelle direction évoluent les adresses [2].



Enfin, l'interaction entre la trace d'exécution et la mémoire est représentée par des flèches. Ces flèches, et en particulier leur entremêlement, permettent de juger du type de séquence d'accès effectués.

1. En cas d'omission de cette flèche, on considérera le sens de lecture habituel. C'est à dire gauche à droite et de haut en bas.
2. La remarque faite précédemment pour le sens de lecture des traces d'exécutions en cas d'omission de la flèche s'applique également pour la mémoire.



4.3 Microbenchmarks

Afin d'évaluer l'impact des interférences sur une cible matérielle donnée, nous souhaitons pouvoir reproduire une multitude de cas de consommation mémoire différents. Si des microbenchmarks destinés à l'étude du système mémoire existent, par exemple STREAM [69], ils ne sont pas adaptés à nos besoins, car destinés à l'évaluation des limites de performances des systèmes mémoires. STREAM, par exemple, génère le trafic le plus intense possible en ne suivant qu'un seul type de séquence d'accès.

Nos microbenchmarks diffèrent des solutions existantes, car ils sont conçus dans l'objectif d'offrir la possibilité de générer des comportements d'accès variés, aussi bien en nature qu'en intensité. À cet effet, nous avons conçu un algorithme générique pour varier les comportements mémoires, dont le pseudocode est donné par l'algorithme 1. Cet algorithme consiste en la répétition d'une séquence d'accès vers une structure de données en mémoire. Cette séquence consiste en trois étapes :

1. Une *boucle de lecture* durant laquelle des données sont lues depuis la mémoire et agrégées dans une variable.
2. Une *boucle de calcul*, dans laquelle la valeur agrégée dans la boucle d'écriture est transformée à l'aide d'opérations arithmétiques simples. Le but de cette boucle est d'inhiber le trafic généré par le microbenchmark en faisant des opérations ne mettant en jeu que des ressources locales à un cœur.
3. Une *boucle d'écriture* où la valeur produite par la boucle de calcul est écrite en mémoire.

Algorithm 1 Microbenchmark

```
function AccessSequence((R, W, C, P_R, P_W))
    for i ← 1 to N do                          ▷ La séquence est répétée N fois
        for j ← 1 to R do                      ▷ Boucle de lecture
            v ← P_R.read(R, pos, v)
            pos ← pos + R
        end for
        for j ← 1 to C do                      ▷ Boucle de calcul
            v ← local_computation(v)
        end for
        for j ← 1 to W do                      ▷ Boucle d'écriture
            P_W.write(W, pos, v)
            pos ← pos + W
        end for
    end for
end function
```

Le comportement des trois boucles composants une séquence d'accès est configurable. Ainsi, les paramètres R, W et C donne respectivement le nombre d'itérations des boucles de lectures, d'écritures et de calcul.

Le paramètre C est directement lié à l'intensité du trafic généré par le microbenchmark, c'est-à-dire la proportion d'instructions générant des accès à la mémoire parmi toutes les instructions exécutées. Cependant, l'effet de C dépends directement des paramètres R et W, car ceux-ci définissent le nombre d'accès à la mémoire effectuée lors d'une séquence d'accès. C'est pourquoi nous utiliserons plutôt le nombre d'itérations de boucle de calcul pour désigner l'intensité supposée du trafic généré.

$$F = \frac{C}{R + W} \tag{3.I}$$

Les paramètres R et W influent à la fois sur la proportion d'accès en lecture et en écriture du trafic généré, mais aussi sur l'entrelacement de ces accès et la longueur des rafales d'accès successifs vers la mémoire. Ceci est illustré par le tableau 4.2 illustrant l'effet des paramètres R et W, pour une valeur de F et un nombre égal de lectures et d'écritures. Ce tableau montre qu'un grand nombre total d'accès ($R + W$) implique à la fois un plus grand nombre d'accès successifs vers la mémoire, mais un plus faible entrelacement des lectures et des écritures. Nous avons pu voir, dans la section 4.1 que ces aspects sont pris en compte par la politique d'ordonnancement du contrôleur

mémoire de notre plateforme matérielle. Notre microbenchmark permet donc de générer des trafics mémoires de même intensité et de mêmes ratios lectures/écritures qui sont néanmoins significativement différents.

TABLE 4.2 – Effet des paramètres R et W sur le trafic généré

R	W	F	C	
1	1	1	2	
3	3	1	6	
6	6	1	12	

4.3.1 Politique d'accès

En plus de leur nombre d'itérations, le comportement des boucles de lectures et d'écritures peut être modifié par le biais de *politique d'accès*, que nous désignons par les paramètres P_R et P_W. Nous avons défini et implanté sept politiques d'accès différentes, dont les caractéristiques sont résumées dans le tableau 4.3. Ces politiques se distinguent notamment par le type de la structure de données accédées, ainsi que par la localité de la séquence des accès.

Structures de données

Nous utilisons trois structures de données différentes :

Tableau Les données sont stockées dans un tableau. Ce sera le cas, quelle que soit la structure de donnée utilisée. Ici, les données sont accédées directement en utilisant l'indexation.

Tableau de pointeurs Les données sont accédées en déférençant un pointeur. Les pointeurs sur les données sont stockés dans un tableau. Ce tableau est parcouru séquentiellement.

Liste chaînée Tout comme pour les tableaux de pointeurs les données sont accédées en déferencant un pointeur. La différence est qu'ici ces pointeurs sont stockés dans une liste chaînée.

Politique	Structure de données	Type d'accès
`linear`	tableau	séquentiel
`random`		aléatoire
`linear-lookup`	tableau de pointeurs	séquentiel
`shuffled-lookup`		séquentiel et aléatoire
`linear-list`	liste chaînée	séquentiel
`shuffled-list`		aléatoire
`stream`	tableaux	parallèle et séquentiel

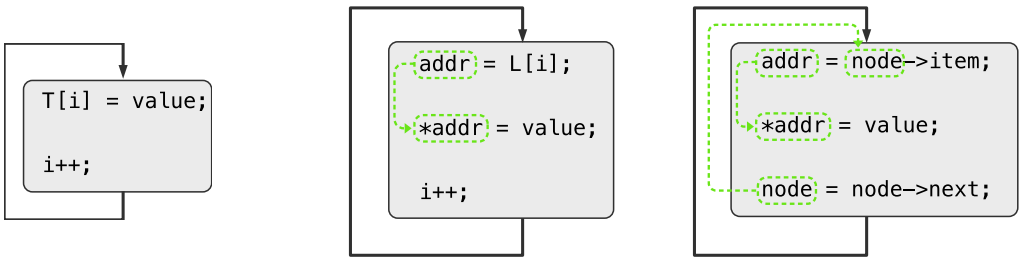
TABLE 4.3 – Récapitulatifs des différentes politiques d'accès

Le choix d'une structure de donnée à des implications sur les dépendances de données présentes dans les boucles de lectures et d'écritures, et plus particulièrement celles concernant les adresses des données accédées lors d'un tour de boucle. Les dépendances de données ont pour effet de donner moins de liberté d'action au processeur pour le réordonnancement d'instructions.

Dans le cas d'un parcours de tableau(figure 4.7a), l'adresse est calculée directement, il n'y a donc pas de dépendances à proprement parler. Il peut y avoir une dépendance entre les tours de boucles pour effectuer ce calcul, mais les valeurs en question étant généralement stockées dans des registres, celle-ci n'est pas significative. L'utilisation d'un tableau de pointeurs (figure 4.7b), introduit une dépendance pour déterminer l'adresse de destination. La conséquence est que le processeur ne peut pas réordonner les instructions au sein d'un même tour de boucle. Par contre, vu que le tableau de pointeur est parcouru séquentiellement, les itérations peuvent l'être. L'utilisation d'une liste chaînée (figure 4.7c) prévient ce degré de liberté en introduisant une dépendance de données entre les tours de boucles.

Politique d'accès

Un deuxième aspect régi par la politique d'accès est la succession d'adresse accédée lors de la séquence. Cela impacte notamment la localité spatiale du trafic émis par le microbenchmark. Les différentes politiques que nous implantons peuvent générer des suites d'adresses pouvant être catégorisées en trois types :



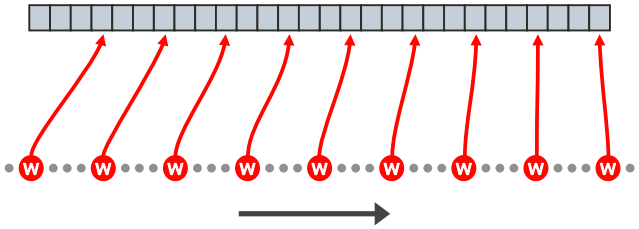
(a) Tableau (b) Tableau de pointeurs (c) Liste chaînée

FIGURE 4.7 – Dépendance de données intra et inter itérations pour les boucles d'accès mémoires en fonction de la structure de donnée utilisée.

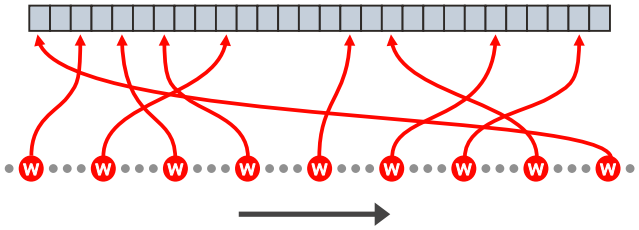
- *Accès séquentiels (ou linéaires)* La différence entre deux adresses successives est constante. On appelle la différence entre deux adresses le *pas (ou stride)*. Il s'agit d'un type d'accès très courant, qui est notamment généré lors l'on parcourt un tableau (figure 4.8a).
- *Accès aléatoires*. Il n'y a pas de relations claires dans l'enchaînement des adresses accédé. Il s'agit du comportement réciproque aux accès séquentiels. Nous utilisons ce type de séquence pour approximer les séquences d'accès les plus complexes (figure 4.8a).
- *Accès parallèle* Plusieurs séquences linéaires sont entrelacées. Ce type de séquence représente en particulier le parcours de plusieurs tableaux en parallèle (figure 4.8a).

La combinaison de ces différents aspects nous permet d'obtenir sept politiques d'accès différentes.

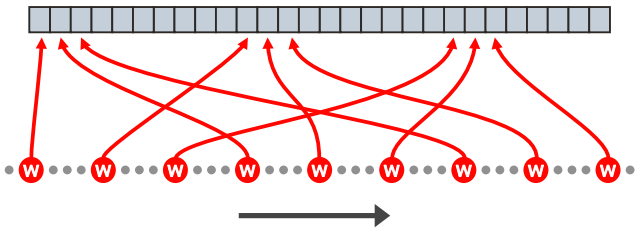
1. La politique `linear` est un parcours séquentiel de tableau. L'adresse accédée lors d'un tour de boucle est donc calculée en ajoutant un pas constant à l'adresse accédée lors de l'itération précédente. Nous employons un pas de 32 octets, correspondant à la taille d'une ligne de cache sur notre plateforme matérielle.
2. La politique `random` implante un parcours de tableau aléatoire. L'adresse accédée est calculée à l'aide d'un générateur de nombre aléatoire. Comme le tableau accède est un tableau d'entier de 32 bits, les adresses sont tout de même alignées sur 4 bits. Afin de limiter le coût de calcul de l'adresse à accéder, nous sacrifions de l'indéterminisme au profit de la performance en utilisant un registre à rétroaction linéaire comme suggéré par Mars et al. [68] et montré dans l'algorithme 4.1.



(a) Accès séquentiels



(b) Accès aléatoires



(c) Accès parallèles

FIGURE 4.8 – Types de séquences d'accès

3.

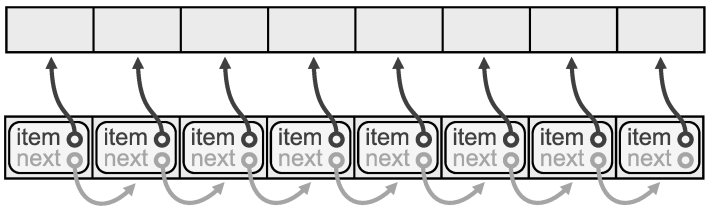
```
#define LFSR_DEFAULT_SEED 0xbadf00d
extern uint32_t lfsr = LFSR_DEFAULT_SEED;

#define MASK 0xd0000001u
#define lfsr_rand() \
((lfsr = (lfsr >> 1u) ^ (uint32_t)(0 - (lfsr & 1u) & MASK)))
```

Listing 4.1 – Générateur de nombre pseudo-aléatoire utilisé dans la politique `random` [68]

La politique `linear-lookup` utilise un tableau de pointeurs pour déterminer quelle adresse accéder. Le tableau est trié et les adresses séparées d'un pas constant, rendant la suite d'adresse finalement accédée séquentielle. La politique `shuffled-lookup` est

similaire, sauf que le tableau de pointeurs est mélangé lors de l'initialisation du microbenchmark afin de rendre la suite d'adresse accédée aléatoire.

La politique `linear-list` utilise une liste chaînée. Les nœuds sont stockés dans un tableau. Ce tableau est trié, un nœud pointe vers le nœud stocké dans la case de tableau adjacente, et deux nœuds adjacents pointent vers des données séparées par un pas constant.



Ainsi le parcours de la liste engendre une suite d'accès linéaire. La politique `shuffled-list` est également similaire, à l'exception que le tableau de nœuds est mélangé à l'initialisation.

La politique `stream` est une généralisation de la partie parallèle de la politique d'accès utilisé par le microbenchmark STREAM [**?**]. Son usage est préconisé pour évaluer la capacité du matériel à traiter des accès en parallèle [23] [96]. Cette politique correspond au parcours séquentiel de plusieurs tableaux en parallèle. Il s'agit d'un type d'accès rencontré fréquemment.

```
offset = 0;
switch(read) {
  case 16:
    value += T[offset + pos];
    offset += ARRAY_SIZE;
  case 15:
    value += T[offset + pos];
    offset += ARRAY_SIZE;
  ...
  ...
  case 1:
    value += T[offset + pos];
  default:
    break;
}
```

Groupe	Comportement	P_R	P_W
MemBench	linear	linear	linear
	random	random	random
	scatter	random	random
	gather	random	random
	linear-lookup	linear-lookup	linear-lookup
	shuffled-lookup	shuffled-lookup	shuffled-lookup
	linear-list	linear-list	linear-list
	shuffled-list	shuffled-list	shuffled-list
Stream	stream	stream	stream

TABLE 4.4 – Comportement d'accès implantés dans nos microbenchmarks

Ces politiques d'accès sont toutes disponibles en lecture et en écriture. Nous appelons *comportement d'accès* une combinaison de politique d'accès en lecture et en écriture. Parmi les 49 combinaisons d'accès possibles nous en implantons 9 divisée en deux groupes et récapitulés dans le tableau 4.4 :

- Le groupe `MemBench` comprend des combinaisons de toutes les politiques d'accès disponible à l'exception de `stream`.
- Le groupe `Stream` ne comprend qu'un comportement qui est une généralisation du microbenchmark STREAM [69].

4.4 Mesures d'interférences

Dans cette section, nous utilisons les microbenchmarks que nous avons définis précédemment pour évaluer l'impact du phénomène d'interférences mémoires sur l'iMX6. Nous avons ici deux objectifs : évaluer la plage de comportement que peuvent produire nos microbenchmarks, et étudier le phénomène des interférences sur l'iMX6. Nous commencerons par détailler le protocole de mesure mis en place, avant de présenter les résultats obtenus.

4.4.1 Protocole de mesure d'interférences

Une mesure d'interférence vise à déterminer le plus grand surcoût temporel observé pour une application. Nous exprimerons ce surcoût temporel en pourcentage du temps d'exécution en isolation T_{iso}. En notant T_{cont} le temps d'exécution en situation de contention, le surcoût temporel est calculé ainsi :

$$Overhead = 100 \cdot \frac{T_{cont} - T_{iso}}{T_{iso}} \tag{4.II}$$

Les temps T_{iso} et T_{cont} sont obtenus expérimentalement, par des mesures de bout en bout. C'est-à-dire que les temps sont déterminés pour l'intégralité d'un chemin d'exécution. Le temps d'exécution T_{iso} est le plus grand temps d'exécution de l'application mesuré face à des programmes *idle*. Le temps d'exécution en contention est le plus grand temps d'exécution de l'application mesuré face à différentes combinaisons de *charges*, qui sont des programmes conçus pour stresser le système mémoire.

Pour que les mesures soient valides, il est important de s'assurer de l'absence de préemption et de migration pour les applications. Afin d'éviter les migrations, les applications sont chacune épinglées à un cœur à l'aide de l'interface POSIX `sched_set_affinity`. Afin d'éviter les préemptions, nous ordonnançons les applications et charges en utilisant la politique temps-réel `SCHED_FIFO` et la priorité maximale. L'utilisation de cette politique avec la priorité maximale est censée garantir l'absence de préemption pour les processus concernés, y compris par des threads noyau. Il faut donc être prudent lorsqu'on utilise cette politique. En effet, si tous les cœurs utilisent cette politique d'ordonnancement pour exécuter des programmes qui ne terminent pas, il devient impossible de reprendre la main. C'est pourquoi, LINUX implante un filet de sécurité pour ce cas précis, au moyen d'un mécanisme dit d'*inhibition temps-réel* ou *RT Throttling*. Ce mécanisme préempte périodiquement les applications ordonnancées avec des politiques temps-réel afin de permettre au noyau de traiter des interruptions. Cette fonctionnalité est activée par défaut, et est une source de perturbations pour nos expériences. Nous l'avons désactivé par le biais du *procfs*.

4.4.2 Ensemble des comportements évalués

À l'aide de nos microbenchmarks, nous constituons un *ensemble de données* pour étudier l'effet des interférences sur notre carte. Cet ensemble de données est composé de mesures (X, y), où X désigne un comportement d'accès à la mémoire et y une mesure de

retards effectuée selon le protocole décrit en section 4.4.1. Nous évaluons une multitude de comportements différents en faisant varier les paramètres de nos microbenchmarks. Ainsi, le comportement d'un microbenchmark est défini par un quadruplet (B, R, W, F) déterminant les paramètres avec lesquelles il est instancié. Le paramètre B désigne un comportement d'accès. Nous évaluons tous les comportements présentés dans le tableau 4.4.

Le paramètre F, rappelons-le, donne le nombre de tours de calcul par boucle d'accès à la mémoire. Nous le faisons varier sur une plage de 0 à 10000 en utilisant une échelle logarithmique, de 0 à 10 F varie avec un pas de 1, de 10 à 100 avec un pas de 10, etc. Plus formellement, l'ensemble des valeurs de F est défini ainsi.

$$T \in \{0, 10000\} \cup \{n \cdot 10^m \mid 0 < n < 10, 0 \leq m \leq 3\} \tag{4.III}$$

Les paramètre R et W donnant les nombres respectifs de tours de boucles de lectures et d'écritures sont générés différemment pour les groupes `Stream` et `MemBench`. En ce qui concerne le groupe *MemBench*, R et W sont définis à partir d'une longueur de rafale BL et d'un ratio de lectures R. Nous testons des rafales d'accès de longueurs 4 et 50, et les ratios de lectures entre 0 et 1 par pas de 0.25.

$$RW_{MemBench} = \{(R \cdot BL, ((1 - R) \cdot BL)) \mid R \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}, BL \in \{4, 50\}\} \tag{4.IV}$$

Pour le groupe `Stream`, les paramètres R et W sont déterminés à partir d'un paramètre mlp, définissant le taux de parallélisme d'accès à la mémoire désirée. Pour une valeur de mlp, on évalue les paires $(1, mlp)$, $(mlp, 1)$ et (mlp, mlp). Nous évaluons les valeurs de mlp 2, 4, 8 et 16.

$$RW_{Stream} = \{(1, mlp), (mlp, 1), (mlp, mlp) \mid mlp \in \{2, 4, 8, 16\}\} \tag{4.V}$$

Résultats globaux

La figure 4.9 montre la relation entre le nombre de tout de boucle de calcul par tour de boucle d'accès (le paramètre F) et le surcoût temporel pour les instances de microbenchmarks constituant le jeu de données défini dans la section 4.4.2. Chaque ligne relie les résultats des instances définies par les mêmes paramètres à l'exception du paramètre F. On dira que ces instances ont une utilisation de la mémoire de *même nature*.

Sur cette figure, on peut tout d'abord observer que le retard global subi décroit exponentiellement avec le paramètre F (figure 4.9a). On peut en effet exprimer la variation du retard O_N subi en fonction de F par l'équation différentielle suivante :

$$\frac{dO_N(F)}{dF} = -\lambda O_N(F) \tag{4.VI}$$

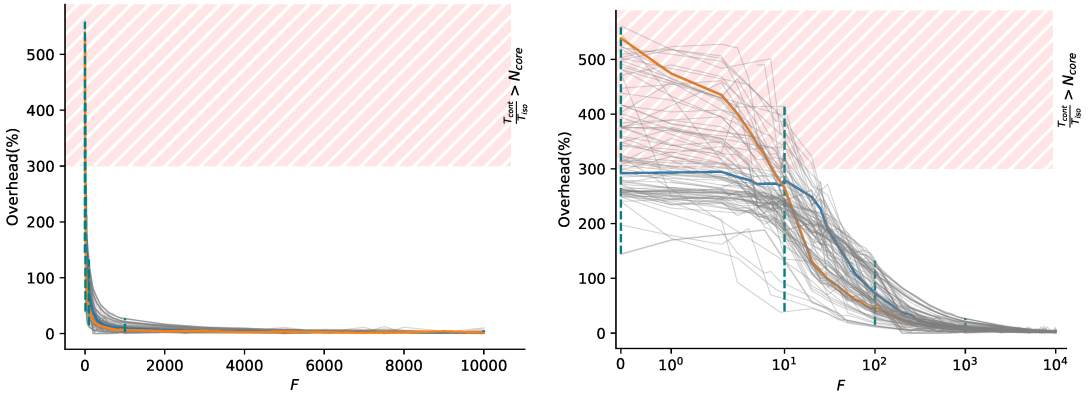
Cette équation signifie que lorsque F augmente, le retard subi diminue proportionnellement avec le paramètre F. La solution de cette équation est

$$O_N(F) = O_N(0) \cdot e^{-\lambda F} \tag{4.VII}$$

On peut donc exprimer la sensibilité d'une nature de trafic en fonction du retard subi pour $F = 0$ (plus grand retard subi) et de la vitesse de décroissance λ. La figure 4.9b utilise une échelle logarithmique en abscisse afin de voir plus en détail la relation entre le pire surcoût temporel observé et F.

Notons d'abord qu'il existe un seuil à partir duquel le surcoût temporel engendré par les interférences est tel qu'il excède le bénéfice que peut apporter le fait d'avoir plusieurs cœurs. Ce seuil est atteint pour une application lorsque le facteur d'inflation de son temps d'exécution excède le nombre de cœurs disponibles. Dans un système avec quatre cœurs, cela correspond à un surcoût temporel de 300%. Nous pouvons constater qu'en pratique ce seuil est non seulement atteint, mais largement dépassé, avec des surcoûts temporels pouvant dépasser les 500%. Parmi les 3870 instances représentées dans la figure 4.9, 233 dépassent le seuil de 300% de surcoût temporel. Le dépassement a lieu pour des trafics avec des intensités élevées, les valeurs de P concernées étant comprises entre 0 et 20.

Les retards observés peuvent non seulement être très importants, ils peuvent également varier énormément pour des valeurs de F pourtant similaires. Ces variations sont mises en évidences sur la figure 4.9 par des lignes verticales montrant l'écart de valeurs observées pour certaines valeurs de F. Les écarts illustrés sont respectivement d'environ 437%, 397%, 123%, 26% et 3% lorsque F est égal à 0, 10, 100, 1000 et 1000. Outre les ordres de grandeur significatifs auxquels nous avons à faire, nous notons que les écarts observés décroissent à mesure que F augmente. Cela indique que la variabilité provient des boucles d'accès à la mémoire.



(a) Échelle linéaire (b) Échelle logarithmique

FIGURE 4.9 – Relation entre le surcoût temporelle et nombre de tours de boucles de calcul par accès à la mémoire

On peut exprimer la sensibilité d'un microbenchmark indépendamment de son paramètre d'intensité au moyen de deux paramètres : le surcoût atteint pour une intensité maximale $O_N(0)$ et le paramètre de décroissance λ. Cela signifie, que l'on ne peut comparer la sensibilité de deux natures de trafic différentes indépendamment du paramètre seulement dans deux cas : celui où elles ont la même intensité maximale et celui où elles décroissent au même rythme. Ceci est illustré par les deux courbes mises en évidence dans la figure 4.9b. Elles se coupent quand $F = 10$, quand $F < 10$ la courbe orange domine très largement la courbe bleue, puis la situation s'inverse. Bien que cela se voit moins quand la courbe bleue domine, les écarts restent significatifs. Par exemple, lorsque $F = 20$, le retard pour la courbe bleue est de 248% et de 131% pour la courbe orange. La distribution des paramètres λ_N et $O_N(0)$ observés en moyenne pour une nature de trafic N est illustré figure 4.10. Elle montre qu'il n'y a a priori pas de lien particulier entre ces deux quantités.

Résultats par type d'applications

Les courbes d'évolutions de retard subi pour les microbenchmarks du groupe `MemBench` sont montrées dans la figure 4.11. On peut y constater que le comportement diffère lorsque les accès sont séquentiels ou aléatoires. Cela se vérifie particulièrement pour les comportements d'accès utilisant des tableaux de pointeurs ou bien des listes chaînées. En effet, ceux-ci permettent de générer des accès aléatoires ou séquentiels sans altérer le comportement du microbenchmark. La différence de retards observée est donc directement imputable à la variation du temps d'accès à la mémoire. Les retards les plus

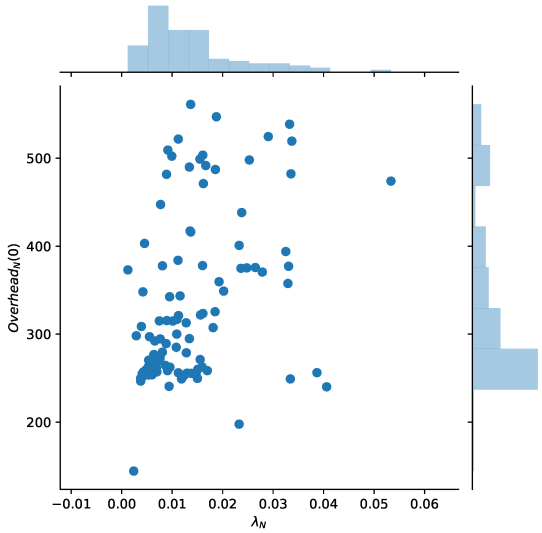


FIGURE 4.10 – Distribution de l'intensité maximale par rapport au facteur de décroissance moyen pour différentes natures de trafic

importants sont observés pour les comportements séquentiels en lecture.

Le comportement des microbenchmarks du groupe `Stream` est montré dans la figure 4.12. Les paramètres importants pour ce groupe sont le stride et le taux de parallélisme des accès à la mémoire mlp. Lorsque $mlp = 2$, le comportement pour un stride donné varie peu. Néanmoins, les retards sont moins importants lorsque le stride vaut 1. En effet, un stride plus faible implique une localité plus élevée et donc moins d'accès vers la mémoire principale. Lorsque le paramètre mlp dépasse 4, on atteint le seuil identifié par Valsan et al. [96] à partir duquel le contrôleur de cache L2 de notre matériel de référence est saturé. Lorsque le stride vaut 8 cela se traduit par une baisse de sensibilité, sauf pour le cas où les lectures sont majoritaires. Ce comportement est en accord avec ce que l'on a pu observer dans le groupe `stream`. Lorsque le stride vaut 1, on observe à la fois une hausse de sensibilité et de la variance. Cela suggère l'importance du rôle du contrôleur de cache L2 dans la sensibilité aux interférences, mais aussi que les cas pathologiques apparaissent plus rarement qu'au niveau de la mémoire principale.

Ces premiers résultats montrent que les interférences peuvent engendrer des retards considérables, représentant souvent plusieurs fois le temps d'exécution des programmes.

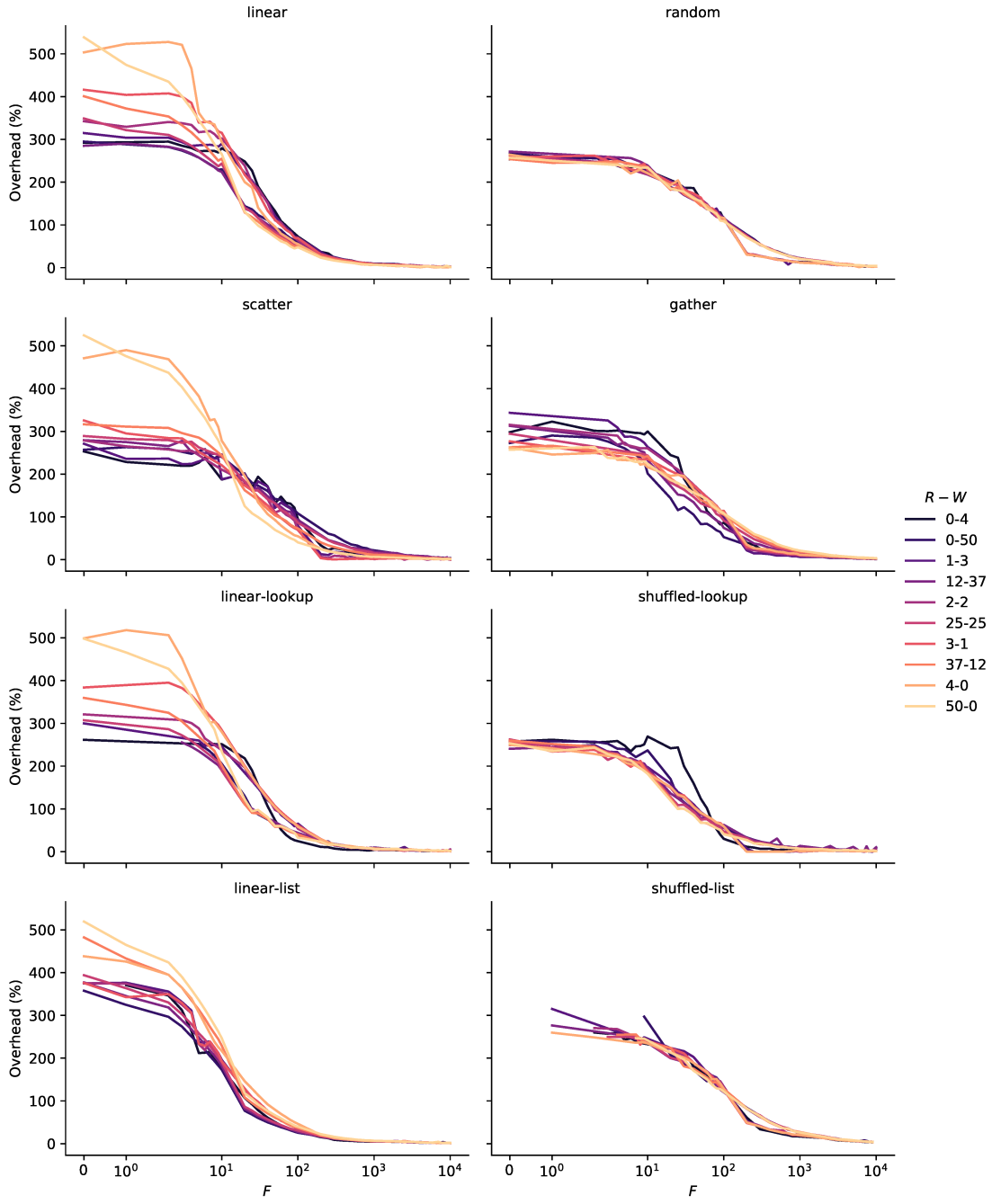


FIGURE 4.11 – Évolution du surcoût temporel avec le nombre de tours de boucle de calcul des microbenchmarks du groupe `MemBench`

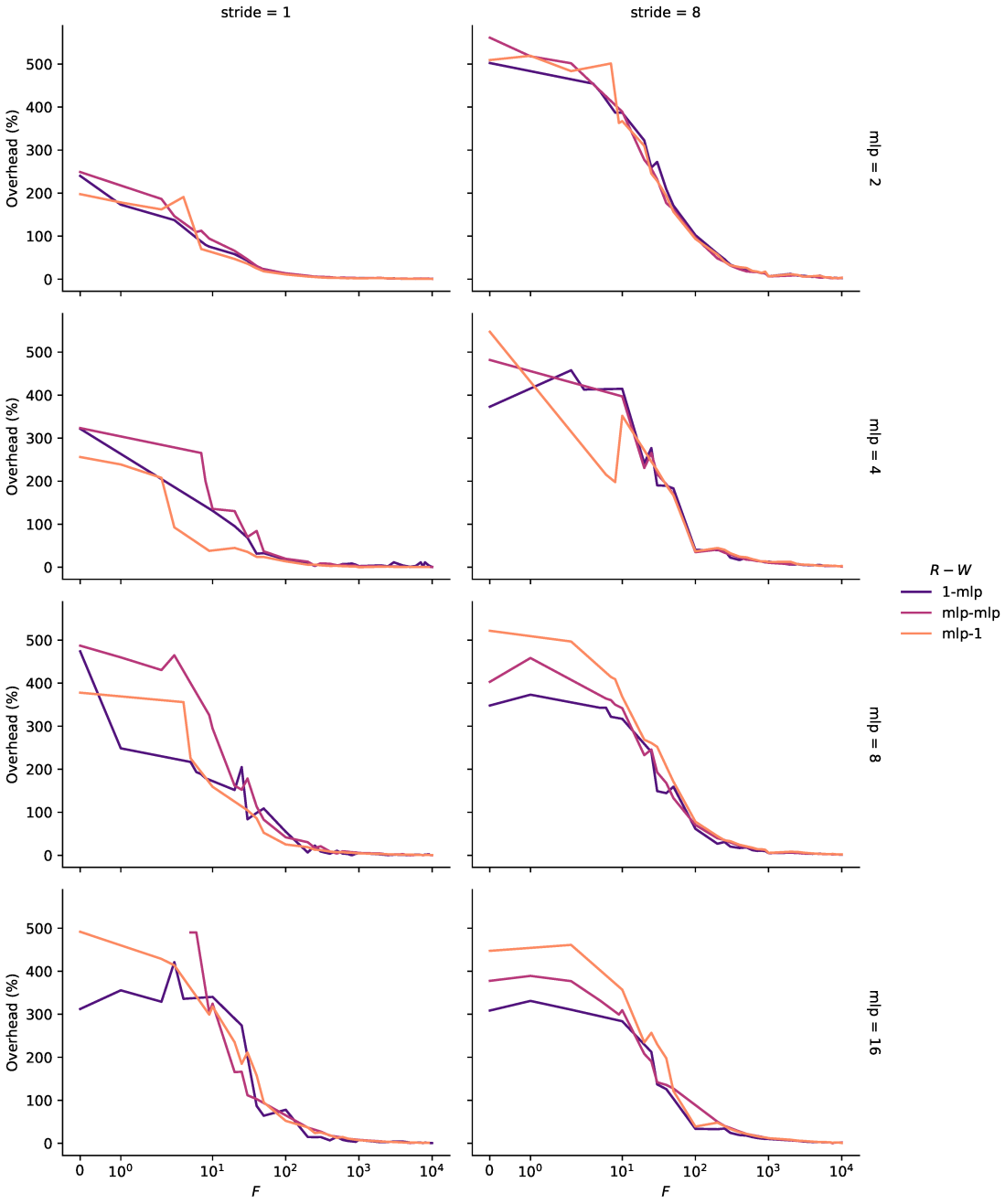


FIGURE 4.12 – Évolution du surcoût temporel avec le nombre de tours de boucle de calcul des microbenchmarks du groupe `MemBench`

4.5 Conclusions

Nous avons, dans ce chapitre, introduit des outils et des méthodes pour l'étude empirique de l'impact des interférences sur une cible multi-cœur COTS. Nous y exposons trois contributions.

Tout d'abord, nous avons introduit une représentation événementielle du comportement d'accès à la mémoire d'un programme. À l'aide de cette représentation, nous avons pu identifier différents aspects caractérisant la nature et l'intensité de l'utilisation de la mémoire que fait un programme. Nous avons, ensuite, développé un ensemble de microbenchmarks permettant de générer une multitude de comportements d'accès à la mémoire, variant selon les aspects que nous avons identifiés précédemment. Enfin, nous avons conduit une étude expérimentale dans le but d'évaluer, d'une part l'ampleur du problème des interférences sur une cible multi-cœur COTS représentative de celle utilisée dans l'industrie, d'autre part le spectre de sensibilités aux interférences offertes par nos microbenchmarks. Cette étude montre que le surcoût induit par les interférences peut être conséquent, avec des facteurs de ralentissements pouvant excéder le nombre de cœurs disponibles. Elle montre également que nos microbenchmarks couvrent un large spectre de sensibilités différentes. Nous notons, plus particulièrement, d'importantes variations causées par des aspects indépendants de l'intensité de l'utilisation de la mémoire.

L'étude expérimentale que nous avons conduite nous a permis de réunir un important ensemble de données sur la sensibilité des applications au phénomène d'interférences. Néanmoins, le comportement des applications dans cet ensemble étant caractérise en fonction des paramètres des microbenchmarks, ces données ne nous donnent pas d'information sur la sensibilité d'applications quelconques. Pour lever cette limitation, nous allons, dans le prochain chapitre, nous intéresser à la mesure de différents du comportement d'accès à la mémoire de manière à pouvoir caractériser le comportement de n'importe quel programme.

CHAPITRE 5

Caractériser les comportements mémoires

Dans le chapitre précédent, nous avons introduit un ensemble de microbenchmarks pour étudier l'impact des interférences sur les applications selon leur comportement d'accès à la mémoire. Nous avons, alors, caractérisé le comportement des microbenchmarks en fonction de leur paramètre. Ces paramètres ne régissant que le comportement de nos microbenchmarks, les résultats de cette étude ne sont donc pas directement transposables à des applications quelconques. Pour remédier à ce problème, nous devons quantifier les aspects pertinents du comportement mémoire en ce qui concerne la sensibilité aux interférences. De notre caractérisation événementielle de l'activité mémoire, nous avons pu distinguer des aspects liés à la nature et à la sensibilité des applications. De cette dichotomie, nous pouvons tirer deux grands groupes de métriques caractérisant l'activité mémoire : les *métriques quantitatives* décrivant les aspects liés à l'intensité du trafic et les *métriques qualitatives* liées à sa nature.

Les métriques quantitatives sont en général mesurables simplement en employant des compteurs matériels de performances. Ce n'est malheureusement pas le cas pour la plupart des métriques qualitatives. Nous contournerons cette limitation en utilisant le modèle événementiel décrit dans le chapitre précédent pour implanter un outil de profilage. Afin de contourner ce problème, nous nous baserons sur le modèle événementiel décrit dans le chapitre précédent pour implanter un outil de capture de profil haute résolu En plus d'être utiles pour mesurer de nouvelles métriques, nous montrerons que cet outil peut être utilisé pour générer des profils en haute résolution de l'activité

mémoire des applications, qui révèlent les différentes phases dans l'exécution de ces dernières.

Le reste de chapitre est organisé de la façon suivante. En premier lieu, nous traitons de la caractérisation quantitative de l'utilisation de la mémoire. En deuxième lieu, nous nous penchons sur la caractérisation des aspects qualitatifs du trafic mémoire. En troisième et dernier lieu, nous présenterons notre solution de profilage.

5.1 Caractérisation quantitative par la bande passante

La caractérisation quantitative décrit la quantité de requêtes d'accès généré par une application. Dans la section 4.4 du chapitre précédent, l'aspect quantitatif du comportement d'accès d'un microbenchmark est contrôlé par le paramètre F, décrivant l'espacement entre les requêtes en lecture et celles en écritures. Pour mesurer l'utilisation quantitative du système mémoire, il est courant d'utiliser la *bande passante*. La bande passante BW d'une application est le nombre d'accès N_{access} effectué sur une durée T.

$$BW = \frac{N_{Access}}{T}$$

Il s'agit d'une grandeur importante pour l'analyse d'interférences, car elle quantifie la fréquence d'accès à la mémoire. Une bande passante élevée impliquant une importante sollicitation des ressources partagées dans le temps, elle indique une grande agressivité sur celles-ci. Paradoxalement, elle indique également une grande sensibilité au problème des interférences. On peut, en effet, directement exprimer le surcoût temporel subi par une application en fonction de sa bande passante. En notant D_{access} le retard moyen subi par accès, on peut exprimer le retard total ainsi :

$$\begin{aligned} D_{total} &= T_{cont} - Tiso \\ &= N_{access} * D_{access} \end{aligned} \tag{1.I}$$

En divisant cette expression par le temps d'exécution en isolation, on peut lier la

bande passante avec le facteur de ralentissement subi.

$$\begin{aligned} \frac{D_{total}}{T_{iso}} &= \frac{T_{cont} - T_{iso}}{T_{iso}} \\ &= \frac{N_{access} * D_{access}}{T} \\ \Leftrightarrow Overhead &= BW \cdot D_{access} \end{aligned} \tag{1.II}$$

La bande passante n'exprime donc pas seulement à quelle fréquence les requêtes d'accès sont émises, mais aussi à quelle fréquence le retard subi en moyenne par chaque accès est propagé dans le temps d'exécution en contention. Le retard subi par une application est donc linéairement croissant en fonction de la bande passante de cette dernière. La vitesse à laquelle le retard croît est donnée par D_{access}. Cette quantité nous permet donc d'étudier le retard subi indépendamment de la consommation mémoire.

Sur les architectures modernes, la bande passante peut se mesurer assez facilement à l'aide de compteurs matériels de performance. La diversité des points de mesures pose néanmoins la question du choix de quel composant de la hiérarchie mémoire considéré pour la mesure. Sur notre plateforme, nous avons identifié deux mémoires sujettes aux interférences : le cache L2 sujet à des interférences temporelles et la mémoire principale sujettes à des interférences spatiales *et* temporelles. Notre carte nous permet de mesurer la bande passante vers ces deux composants :

- La bande passante vers la mémoire principale (notée BW^{DRAM}) est mesurée à l'aide de la PMU situé dans le $MMDC$ de notre carte.
- La bande passante vers le cache L2 (notée BW^{L2}) est mesurée à l'aide du compteur de défauts de cache L1 fourni par la PMU du cœur exécutant l'application.

Le partage ou non des points de mesures ont une influence sur ce que l'on peut mesurer. La bande passante vers la mémoire principale se mesure à l'aide d'une PMU situé dans le contrôleur mémoire, dont les compteurs enregistrent le trafic émis par tous les cœurs en même temps. On ne peut donc pas quantifier la part individuelle de chaque cœur dans le trafic enregistré. La bande passante vers la mémoire principale ne peut donc être enregistrée qu'en isolation. Ce qui n'est pas un problème pour la caractérisation de comportement. Cette limitation n'existe pas pour la bande passante vers le cache L2. Dans tous les cas, lorsque nous parlerons de bande passante il sera sous-entendu bande passante *en isolation*.

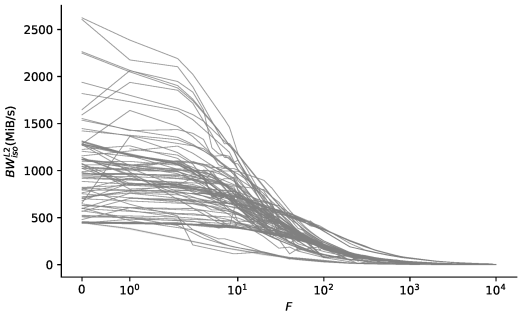
Prise séparément, chacune de ces bandes passantes ne peut exprimer qu'une portion du trafic au sein du système mémoire. Dans le cas de la mémoire principale, la bande passante vers ce composant est une mesure du débit d'accès ayant traversé toute la hiérarchie mémoire. Une incertitude demeure sur les accès n'ayant traversé que partiellement la hiérarchie. Sur notre carte, c'est le cas des défauts de cache L1 n'ayant pas entraîné de défaut de cache L2. Réciproquement, la bande passante vers le cache L2 ne capture pas l'activité pour les composants situés après celui-ci. Cette incertitude est une limitation pour les systèmes de régulation basés sur la bande passante. MemGuard [108] utilise la bande passante vers le cache L2, tandis que Blin et al. [24] utilisent la bande passante vers la mémoire principale.

Étude expérimentale

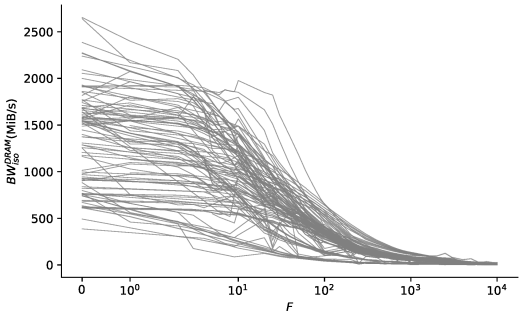
La bande passante étant facile à mesurer, nous avons étendu l'analyse d'interférences présentée dans le chapitre précédent afin d'étudier la relation qu'elle avec le retard subi par une application. Nous utilisons dans cette étude le même jeu de donnée que dans le chapitre précédent.

Les figures 5.1b et 5.1a illustre la relation entre le paramètre F quantifiant le nombre de tours de boucle de calculs par accès mémoire et les différentes bandes passantes produites. Qu'il soit mesuré par la bande passante produite vers la mémoire principale ou bien celle produite vers le cache L2, le trafic généré décroît exponentiellement. L'étendue des bandes passantes observée en $F = 0$ est similaire pour BW_{iso}^{DRAM} et BW_{iso}^{L2}, elle est comprise entre 500 et 2700 MiB/s. Notons cependant une disparité dans la distribution de ces bandes passantes, la majorité des bandes passantes vers le cache L2 (pour $F = 0$) sont relativement faibles, tandis que celle vers la mémoire principale est répartie plus uniformément. Cela indique que la bande passante vers la mémoire principale est généralement plus élevée que celle vers le cache L2. La figure 5.1c illustre la relation entre la bande passante vers la mémoire principale et celle vers le cache L2. La ligne verte correspond au cas où les bandes passantes sont égales, et la ligne rouge à celui où la bande passante vers la mémoire principale est deux fois supérieure à celle vers le cache L2. On constate que la majorité des points sont compris entre ces deux lignes. D'une part, cela indique qu'en général, il n'y a pas de défaut de cache L1 entre les défauts de cache L2 dans nos microbenchmarks, la majorité des points étant sur ou sous la ligne verte. D'autre part, cela indique que plus d'une ligne de cache est chargée à la fois lors des accès vers la mémoire principale.

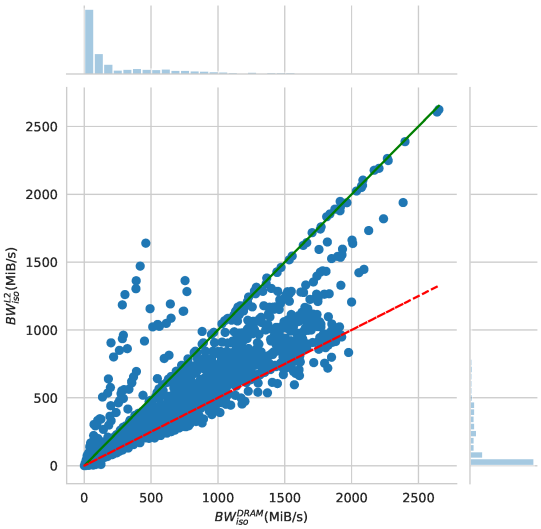
La relation entre la bande passante et le retard subi est illustrée figure 5.2. Pour les



(a) Bande passante vers le cache L2 en fonction de F



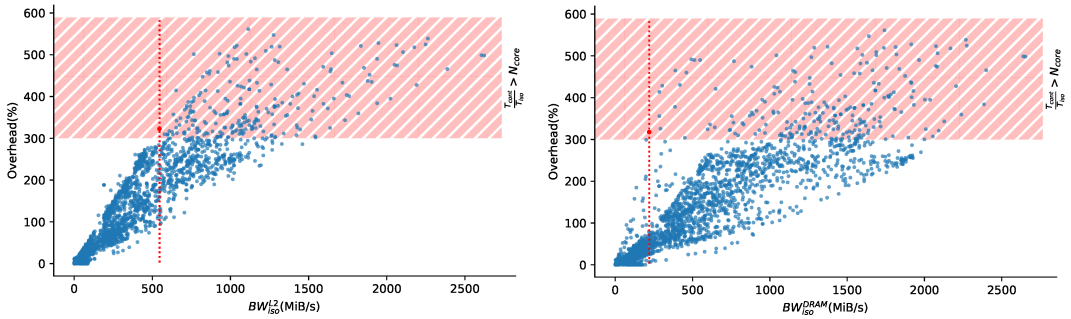
(b) Bande passante vers la mémoire principale en fonction de F



(c) Bande passante vers le cache L2 en fonction de la bande passante vers la mémoire principale

FIGURE 5.1 – Bande passante de l'ensemble de données synthétique

deux types de bande passante étudiée, le facteur de ralentissement subi croit avec la bande passante. On peut également observer une forte dispersion des retards subie pour des bandes passantes similaires. En particulier, si nous n'observons pas d'application avec une bande passante élevée et un retard faible, la réciproque se révèle fausse. Le seuil où le facteur de ralentissement subi est plus important de nombre de cœurs est dépassé pour des bandes passantes d'environ 550 MiB/s pour BW_{iso}^{L2} et 200 MiB/s pour BW_{iso}^{DRAM}, soit respectivement moins de 25% et de 10% de la bande passante atteignable. On observe une plus grande dispersion de valeur pour la bande passante vers la mémoire principale, en particulier pour les bandes passantes faibles, indiquant des valeurs de D_{access} plus élevées lorsque calculée à partir de cette bande passante.



(a) Bande passante vers le cache L2 (b) Bande passante vers la mémoire principale

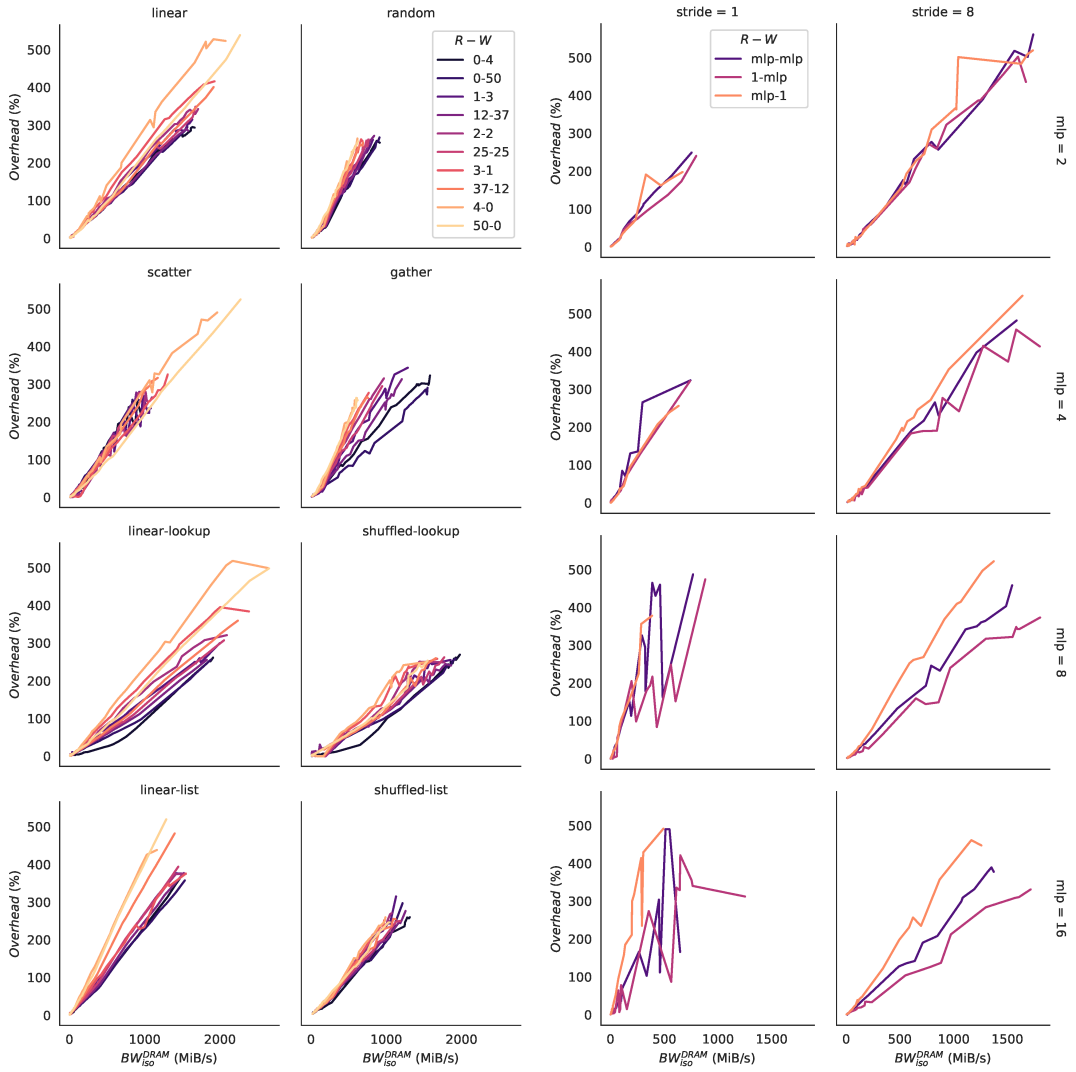
FIGURE 5.2 – Facteur de ralentissement subi en fonction de la bande passante en isolation

L'évolution du retard subi en fonction de la bande passante vers la mémoire principale est montrée par nature de trafic dans la figure 5.3. On retrouve la relation linéaire exprimée dans l'équation 1.II. Cela signifie que pour nature de trafic le retard moyen par accès D_{access}^{DRAM} est relativement constant. On peut néanmoins observer de légères variations pour les différentes courbes, que l'on peut imputer à la mesure. Les variations les plus importantes s'observent pour les applications du groupe stream, en particulier celle avec un stride de 1.

Les pentes des différentes courbes de la figure 5.3 sont variées, les valeurs de D_{access}^{DRAM} pouvant dépasser les 500 cycles. Si on applique la formule 1.II avec la plus grande valeur de D_{access}^{DRAM} observé en (302,86 cycles par accès) pour une nature de trafic donnée et la plus grande bande passante observée (0, 086 accès par cycles), le retard obtenus est de 2617,6%, soit un facteur de ralentissement de 27.

Fort heureusement, en étudiant la distribution du retard moyen par accès en fonction la bande passante (figure 5.4), on peut estimer que la probabilité d'associer une bande passante élevée à une pente élevée est faible. Observons, tout d'abord, la relation entre le retard moyen observé par accès et la bande passante vers la mémoire principale (figure 5.4a). Non seulement les valeurs extrêmes de retard par accès ne s'observent que pour les bandes passantes faibles, mais, de plus, ces valeurs décroissent quand la bande passante augmente.

Les valeurs extrêmes de D_{access}^{DRAM} peuvent en partie s'expliquer par la contention sur le contrôleur de cache L2. À cette fin, la figure 5.4b montre la relation entre le retard moyen par accès observé *en moyenne sur les instances de microbenchmarks caractérisant la même nature de trafic* et le nombre d'accès vers le cache L2 par accès vers la mémoire

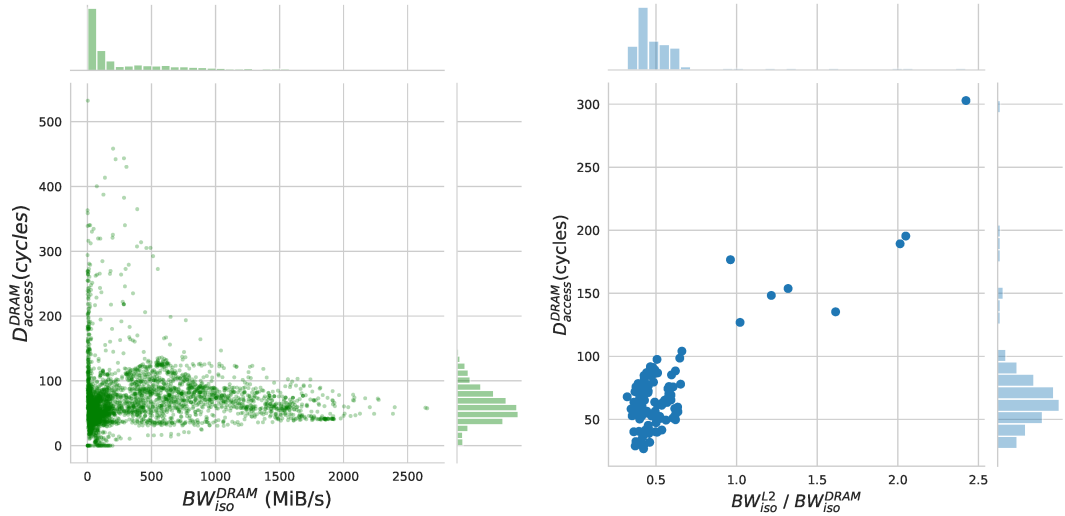


(a) `MemBench` (b) `Stream`

FIGURE 5.3 – Facteur de ralentissement subi en fonction de la bande passante en isolation par nature de trafic généré

principale. On y distingue deux zones, la première correspond aux cas où le cache L2 n'est pas sollicité entre les accès (c'est-à-dire lorsque la valeur en abscisse est inférieure à 1), la deuxième regroupe les cas où il l'est. On constate que les valeurs extrêmes sont exclusivement dans cette deuxième zone. De plus, il y a, dans cette zone, une corrélation entre le retard moyen par accès et le nombre d'accès vers le cache L2 par accès vers la mémoire principale. Cela indique clairement que la contention sur le cache L2 peut jouer un rôle important sur le retard subi. Il convient tout de même de relativiser cet effet, un

nombre de défauts de cache L1 important entre les défauts de cache L2 impliquant un espacement des accès vers la mémoire principale plus important, et donc une bande passante plus faible. Cela nous donne une raison supplémentaire, de penser qu'il est improbable d'observer des retards par accès et des bandes passantes élevées en même temps. Si la contention sur le cache L2 explique une partie de la variation dans les pentes observée, d'importantes variations demeurent. En effet, l'examen de la zone de la figure 5.4b correspondant à l'absence de sollicitation du cache L2 entre les accès vers la mémoire principale montre des valeurs de retard variant du simple au quadruple (entre 25 et 100 cycles).



(a) Relation entre la bande passante et le retard par accès (un point représente une instance de microbenchmark)

(b) Relation entre le nombre d'accès vers le cache L2 par accès vers la DRAM et le retard moyen par accès (un point représente la valeur moyenne pour une nature de trafic)

FIGURE 5.4 – Distribution de D_{access}^{DRAM} par rapport à la bsande passante et au nombre d'accès vers le cache L2 par accès à la mémoire principale

5.2 Quantifier les aspects qualitatifs

Nous avons, dans la section précédente, étudié le lien entre la bande passante d'une application et le surcoût temporel qu'elle subit à cause de la contention mémoire. De cette étude, nous pouvons conclure que la bande passante est une caractéristique importante de la sensibilité d'une application aux interférences, mais qu'elle n'explique pas tout. En effet, les importantes variations de retard observé que l'on peut observer pour des bandes passantes analogues laissent à penser que la nature du trafic joue aussi un rôle

dans la sensibilité aux interférences. Dans cette section, nous présentons des métriques mesurant certains aspects qualitatifs du trafic mémoire.

5.2.1 Proportions de lectures et d'écritures

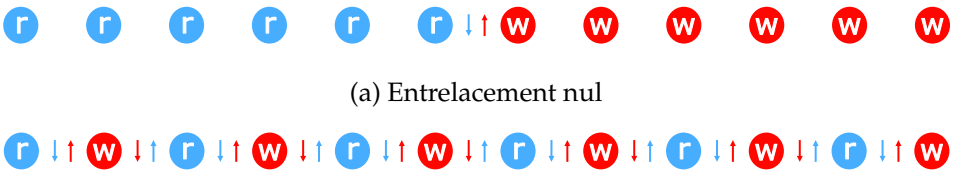
Un premier aspect qualitatif du trafic mémoire est la proportion de requêtes en lecture et en écriture parmi les accès qui le composent. Les lectures et les écritures sont affectées de différentes façons par le problème des interférences. Par exemple, sur notre cible matérielle, les requêtes en lectures et en écritures ont une file dédiée en fonction de leur type dont la taille diffère. Les requêtes sont également traitées différemment en fonction de leur type [44]. De plus, par leur sémantique, les lectures et les écritures n'affectent pas les progrès des applications de la même manière. Les lectures sont synchrones à moins d'être causées par des instructions de préchargement. C'est-à-dire qu'à un moment donné le progrès du programme va dépendre de l'arrivée d'une donnée. Réciproquement, les écritures sont à priori asynchrones, et donc à priori moins susceptibles de bloquer l'exécution du programme.

Nous mesurons cette quantité à l'aide de la PMU situé dans le MMDC de notre carte. Nous l'exprimons par le nombre d'accès en lecture par rapport au nombre d'accès total.

$$RW = \frac{N_{lectures}}{N_{acces}}$$

5.2.2 Entrelacement des lectures et des écritures

Lorsque le trafic comporte à la fois des lectures et des écritures, la répartition de ces types d'accès peut varier. L'entrelacement des lectures et des écritures peut s'exprimer en comptant le nombre de fois que le sens d'accès vers la mémoire varie. Par exemple, si le sens d'accès à la mémoire ne varie qu'une fois alors tout les accès d'un type sont suivis par tous les accès de l'autre type (figure 5.5a). Réciproquement, si le sens change à chaque accès, alors les lectures et les écritures sont complètement entrelacées (figure 5.5b). Cet entrelacement peut avoir différents impacts, aussi bien sur le progrès de l'application que sur la dégradation du temps d'accès à la mémoire.



(a) Entrelacement nul

(b) Entrelacement complet

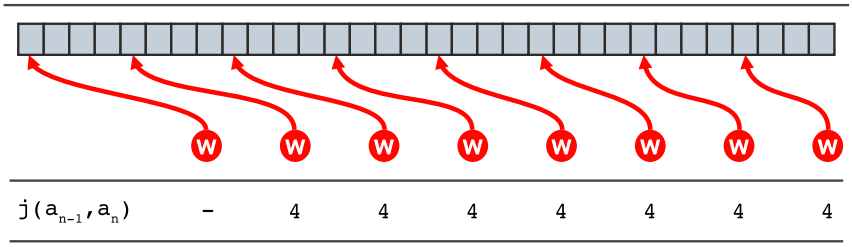
FIGURE 5.5 – Inversions de la direction d'accès à la mémoire en fonction du niveau d'entrelacement

En ce qui concerne le progrès des applications, des lectures et d' écritures très entrelacées peuvent (mais pas toujours) induire plus de dépendances de données. Ces dépendances peuvent alors bloquer l'exécution du programme le temps que des données nécessaires soient chargées. Elles peuvent aussi laisser moins de possibilités au processeur pour réordonnancer des instructions, et donc d'éventuellement compenser la dégradation du temps de service des accès mémoire.

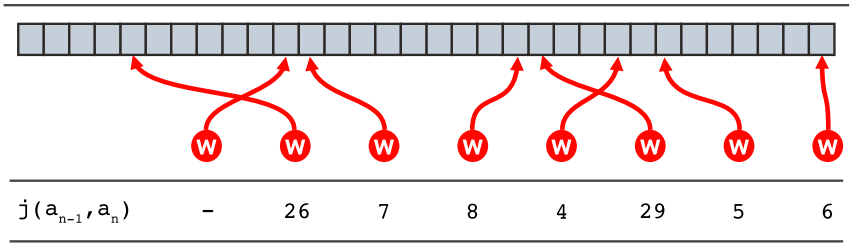
Dans la section 4.1.5, nous avons vu que le passage d'un type de requête à l'autre a un impact sur le temps de service des accès à la mémoire principale. Le bus de donnée de la DRAM étant unidirectionnel, il doit être inversé pour passé de lecture à écriture ou réciproquement. Cette inversion du bus de données est coûteuse, car elle rend la mémoire principale indisponible pendant qu'elle se fait. Sur le matériel que nous utilisons, les requêtes causant une inversion du bus de données sont pénalisées par l'algorithme de réordonnancement de requêtes implantées dans le MMDC. Les applications générant du trafic avec des lectures et des écritures très entrelacées sont donc plus susceptibles de souffrir de l'iniquité du processus de réordonnancement, et donc d'être plus sensible au problème des interférences.

5.2.3 Complexité des séquences d'accès

La séquence d'accès d'un programme définit comment les accès s'enchaînent d'un emplacement mémoire à l'autre. La figure 5.6 illustre deux exemples de séquences d'accès, l'une est séquentielle, c'est-à-dire que la mémoire est accédée à pas constant, l'autre est aléatoire. La différence de structure dans l'entremêlement des flèches traduit une différence de complexité entre ces deux séquences. La séquence linéaire est, en effet, plus simple à représenter que la séquence aléatoire. Nous voulons quantifier cette complexité.



(a) Accès séquentiels



(b) Accès aléatoires

FIGURE 5.6 – Sauts entre adresses de 5 bits

Pour des adresses dont la taille est b bits, on définit le saut j d'une adresse a_1 à une adresse a_2 comme ceci :

$$j(a_1, a_2) = (a_2 - a_1) \bmod 2^b$$

Nous avons fait le choix de calculer les sauts modulo le nombre d'adresses totales de l'espace d'adressage. Ce choix nous permet notamment d'assurer qu'un parcours séquentiel, génère une séquence de sauts constante, y compris en cas de dépassement d'adresse. Cette propriété peut sembler anecdotique dans le cas d'adresse de 32 bits ou plus, mais les dépassements sont fréquents lorsque l'on étudie les sauts entre des portions d'adresses, par exemple, les bits d'indexation utilisés par le cache. La contrepartie de mode de calcul est que les sauts sont toujours positifs. Les sauts en arrière sont représentés par de longs sauts en avant, causant un débordement d'adresse. Un exemple de sauts vers l'arrière est donné dans la figure 5.6b.

Afin de quantifier la complexité d'une suite de sauts, nous nous appuyons sur le concept d'*information propre* de la théorie de l'information [89]. L'information propre I d'un saut j appartenant à une suite d'accès P est une mesure de la quantité d'information apportée par j pour déduire l'enchaînement complet. Elle se base sur la probabilité $p(j)$

d'occurrence de j dans P.

$$I(j) = -log(p(j)) \tag{2.III}$$

Le principe de l'information propre est qu'un événement arrivant souvent ne donne pas beaucoup d'informations. Par exemple, dans le cas d'une suite d'accès séquentiels, tous les sauts sont identiques. Clairement, il suffit d'observer un seul saut pour en déduire tout le reste de la séquence, et donc faire d'autres observations ne nous donnera pas plus d'informations. Dans ce cas précis, l'information propre de tous les éléments de l'enchaînement d'accès vaut 0 vu que $p(j) = 1$, et donc $I(j) = -log(1) = 0$. L'*entropie de Shannon* d'une séquence est l'information propre moyenne des éléments d'une séquence d'accès. Il s'agit du nombre de bits nécessaire pour encoder l'enchaînement complet.

$$H(P) = \mathbb{E}[I(P)] = -\sum_{j \in P} p(j) log(p(j)) \tag{2.IV}$$

L'entropie de Shannon est une mesure de la complexité d'un enchaînement d'accès. Des programmes avec une entropie élevée ont de fortes chances d'avoir une mauvaise localité spatiale, et par conséquent une plus grande sollicitation de composants du système mémoire telle que les lignes de DRAM. Il s'agit aussi d'une mesure de la difficulté pour les mécanismes de spéculations de remplir correctement leur rôle, et par extension réduire l'impact des interférences. En effet, on peut interpréter l'information propre comme un coût pour prédire correctement les accès futurs à partir de l'historique des accès passés.

Pour atteindre l'entropie maximale, il faut effectuer suffisamment d'accès. Dans le cas de l'enchaînement d'adresse de 32 bits cela représente 2^32 accès, ce qui n'est pas toujours atteint. Afin de résoudre ce problème, nous utilisons une forme normalisée de l'entropie. Si les adresses qui s'enchaînent sont encodées sur b bits, elle s'exprime ainsi :

$$\overline{H}(P) = \frac{H(P)}{min(log_2(N_{access}), b)} \tag{2.V}$$

5.2.4 Impact du temps de service des accès mémoire sur la vitesse d'exécution

La nature de la consommation mémoire d'un programme a un impact sur le temps de service des requêtes d'accès. Par exemple, sur notre plateforme matérielle, des aspects qualitatifs impactent le comportement du contrôleur DDR. Cela peut également être le cas pour d'autres composants de la hiérarchie mémoire. En pratique, ce temps de service peut être plus ou moins répercuté sur la vitesse de progression de l'application. Les instructions causant des accès vers la mémoire principale ont un temps d'exécution considérablement plus important que les autres. Vu que l'intensité du trafic I mémoire est le nombre d'accès par instructions, elle détermine largement le temps T_{mem} passé par l'application à faire des accès mémoire, et le temps T_{comp} passé à exécuter d'autres instructions. Ainsi, l'intensité a un fort impact sur la vitesse de progression de l'application, mesurée en nombre de cycles par instructions, notée CPI. Le CPI d'une application peut être exprimé comme la combinaison linéaire du nombre de cycles moyen pour accéder à la mémoire CPI_{mem} et le nombre de cycles moyen pour exécuter les autres instructions CPI_{comp}.

$$\begin{aligned} CPI &= \frac{T}{N_{inst}} \\ &= \frac{T_{mem} + T_{comp}}{N_{inst}} \\ &= \frac{N_{access} \cdot CPI_{mem} + (N_{inst} - N_{access}) \cdot CPI_{comp}}{N_{inst}} \\ &= \frac{N_{access}}{N_{inst}} \cdot CPI_{mem} + \frac{(N_{inst} - N_{access})}{N_{inst}} \cdot CPI_{comp} \\ &= I \cdot CPI_{mem} + (1 - I) \cdot CPI_{comp} \end{aligned} \tag{2.VI}$$

La valeur de CPI_{mem} est très dépendante du temps T_serv mis par le contrôleur mémoire pour servir les accès. Néanmoins T_{serv} n'est que rarement répercuté exactement sur la valeur de CPI_{mem}. Cela peut s'expliquer par des raisons architecturales. Pour offrir de bonnes performances, notre plateforme matérielle utilise des fonctionnalités comme les pipelines, des caches, de l'exécution spéculative et dans le désordre, etc. Ces fonctionnalités ont été identifiées comme pouvant entraîner des anomalies temporelles et des effets dominos, rendant l'architecture de notre plateforme non compositionnelle. Cela signifie que le temps total par accès CPI_{mem} n'est pas égal à la somme des temps de service T_{serv} des différents composants du système mémoire. Nous faisons l'hypothèse

que le temps de service des accès au niveau de la mémoire principale constitue la part la plus importante du temps de service total et exprimons la différence dans le temps d'accès total ainsi :

$$CPI_{mem} = \tau \cdot T_{serv} \tag{2.VII}$$

Le facteur τ mesure donc comment le temps de service de la mémoire principale est répercuté sur le temps total d'accès à la mémoire. En injectant cette définition dans l'équation, 2.VI on peut calculer ce facteur de la façon suivante :

$$\tau = \frac{CPI - (1 - I) \cdot CPI_{comp}}{I \cdot T_{serv}} \tag{2.VIII}$$

Nous mesurons T_{serv} en utilisant la PMU du contrôleur mémoire de notre plateforme. L'intensité du trafic est mesurée à l'aide du nombre d'accès mesurés sur la PMU du contrôleur mémoire et le nombre d'instructions est mesuré sur la PMU locale du cœur exécutant l'application. La valeur de CPI_{comp} étant difficile à mesurer, nous utilisons une approche conservative et utilisons la valeur de CPI la plus basse atteignable sur notre cible, ici 0,25.

5.2.5 Récapitulatif des métriques qualitatives

Nous récapitulons, ici, les différentes métriques qualitatives définies dans cette section.

La *proportion de lecture/écritures*, notée RW, mesure la proportion de requêtes d'accès en lecture dans le trafic total. Cette métrique quantifie la proportion de lectures et d'écritures dans le trafic généré par l'application. Nous pouvons mesurer cette quantité directement sur notre plateforme matérielle. Nous utilisons pour cela les compteurs situés dans le contrôleur DRAM, ce dernier permettant de distinguer le trafic en lecture du trafic en écriture.

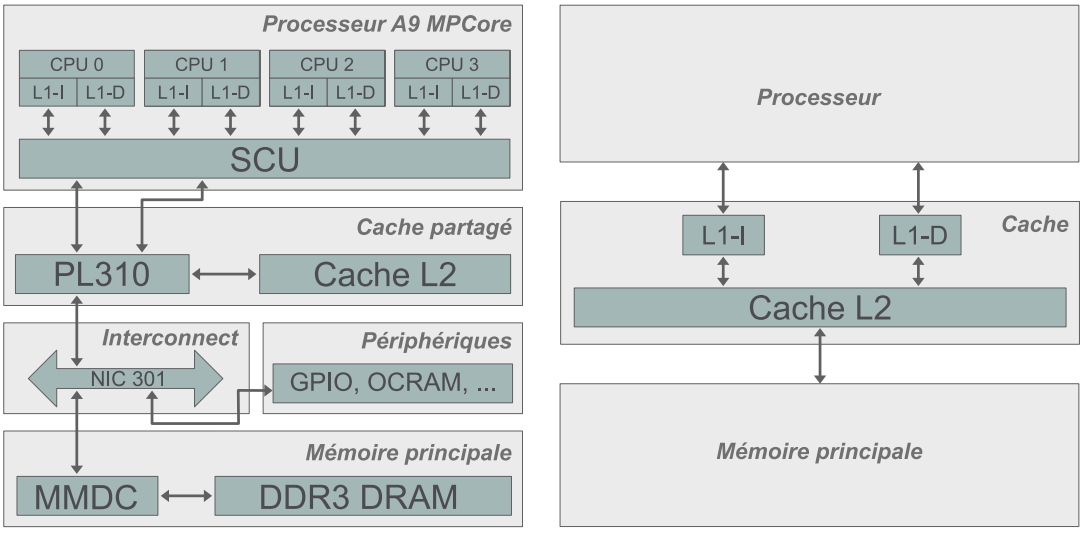
Le *taux d'entrelacement des lectures et des écritures*, noté DBI, est une mesure la fréquence à laquelle le sens d'accès à la mémoire s'inverse. Pour mesurer cette métrique, il faut disposer d'un compteur indiquant le nombre d'alternances du sens d'accès à la mémoire. Notre plateforme ne disposant pas d'un tel compteur, nous ne pouvons mesurer ce taux directement.

L'*entropie des séquences d'accès*, notée H, mesure la complexité des séquences d'accès vers la mémoire. C'est la quantité d'information moyenne requise pour reconstituer une séquence d'accès vers la mémoire. La mesure de cette quantité requiert une analyse fine des accès effectués vers la mémoire, que ne nous permet pas notre plateforme matérielle. Nous ne pouvons donc pas la mesurer directement.

Le *facteur* τ mesure l'impact du temps de service des requêtes d'accès DRAM sur la vitesse d'exécution du programme. Pour calculer ce facteur, il faut pouvoir mesurer la vitesse d'exécution du programme, ainsi que le temps de service moyen des accès DRAM. Notre plateforme matérielle nous permet de mesurer ces deux métriques. La première à l'aide du compteur d'instruction situé dans les cœurs de calcul, la seconde à l'aide des compteurs de cycles et de cycles occupés situés dans le contrôleur DRAM.

5.3 Profilage de l'activité mémoire

Nous venons de voir que certaines métriques qualitatives que nous avons décrites dans la section précédente ne sont pas mesurables à l'aide de compteurs de performances. Afin de les mesurer, nous avons donc recours à une forme de simulation. Le problème qui se pose est que le matériel disponible dans le commerce et destiné à des applications grand public présente généralement deux caractéristiques : l'opacité et la complexité. Afin de contourner ce problème, nous nous attachons à la représentation événementielle que nous avons utilisée jusqu'à présent, c'est-à-dire un flux de requêtes d'accès émises vers un système mémoire partagé *boite noire*. Dans notre configuration, la mémoire partagée sujette aux interférences spatiales est la DRAM. Nous avons fait le choix d'inclure dans la boite noire la DRAM ainsi que l'interface vers cette dernière. Les accès mémoires sont donc émis lors des défauts de cache de niveau 2.



(a) Modèle complet (b) Modèle utilisé par CACHEGRIND

FIGURE 5.7 – Vues de l'architecture matérielle

Nous avons développé un outil pour capturer le trafic explicitement généré lors de l'exécution d'un programme. Il est basé sur l'outil CacheGrind du cadriciel d'instrumentation binaire dynamique VALGRIND. CacheGrind est un outil de profilage de cache. Nous utilisons le simulateur de cache de cet outil pour déterminer le flux de requêtes d'accès vers la mémoire partagée. Comme illustré sur la figure 5.7, hormis la hiérarchie de caches, le matériel est considéré comme une boite noire et n'est donc pas simulé. Le choix de VALGRIND est motivé par la maturité du projet et le nombre de jeux d'instructions supportés.

Le processus de profilage, illustré dans la figure 5.8, est le suivant :

1. *Compilation* Le profileur fonctionnant à partir du binaire, il ne s'agit pas à proprement parler d'une étape de profilage. Notons cependant qu'aucune option de compilation particulière n'est requise.
2. *Émulation* Le flux de requête d'accès est émulé à l'aide d'un outil VALGRIND
3. *Conversion* Le trafic émulé est enregistré dans un format de profils haute résolution.
4. *Représentation* Les données d'un profil en haute résolution sont exploitées de deux manières différentes.
   a) *Visualisation* Le profil peut être converti en image pour la visualisation.
   b) *Agrégation* Les données peuvent être utilisée pour le calcul de métriques.

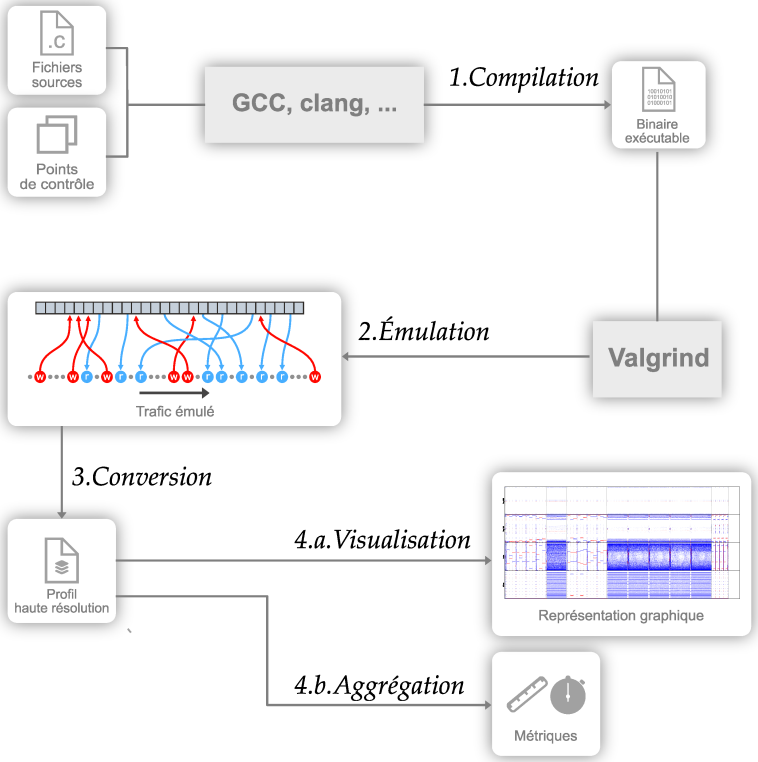


FIGURE 5.8 – Processus de profilage

5.3.1 Émulation du trafic

Nous nous reposons sur VALGRIND pour l'émulation du trafic généré par une application. VALGRIND est un cadriciel d'instrumentation binaire dynamique, il permet d'instrumenter le code d'un binaire lors de son exécution. VALGRIND est constitué de deux parties :

- La partie *valgrind core* fournit des services pour le parcours du graphe de flot de contrôle de l'application.
- La partie *outil* définit le type d'instrumentation du binaire.

Exécution d'un programme par Valgrind

VALGRIND est un framework d'instrumentation binaire dynamique, sa fonction principale est d'instrumenter des blocs de code à la volée. Ainsi, lorsqu'un programme est exécuté dans VALGRIND, les blocs de bases du programme sont instrumentés au fur et à mesure du parcours du graphe de flot de contrôle pour une entrée donnée.

L'exécution d'un bloc de base du programme implique les étapes montrées dans la figure 5.9 :

1. L'étape de *traduction* consiste à convertir le bloc de base de sa représentation binaire vers une représentation intermédiaire nommée *VEX IR*. Elle est assurée par la partie *valgrind core*.
2. Le bloc converti en représentation intermédiaire est ensuite passé à la partie *outil* pour l'étape d'*instrumentation*. Un outil définit donc le traitement a apporté à un bloc de base. La phase d'instrumentation est également l'occasion de procéder à l'*analyse* des blocs passés en entrée. Notons cependant que cette analyse se fait sur le bloc en représentation intermédiaire.
3. Une fois instrumenté, le bloc peut être exécuté. Pour cela, la partie *valgrind core* compile celui-ci en code natif.

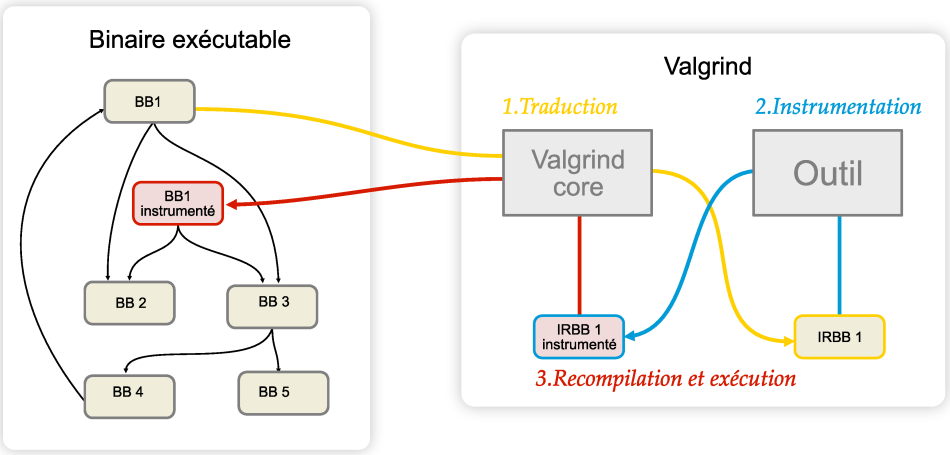


FIGURE 5.9 – Émulation d'un bloc de bases avec Valgrind

L'outil CACHEGRIND

Un outil VALGRIND définit comment les blocs de bases sont instrumentés. En pratique, ils sont plutôt destinés à l'analyse de programme. Le plus connu (et celui utilisé par défaut), MEMCHECK [4], servant à détecter les fuites mémoires. Nos travaux reposent sur l'outil CACHEGRIND [2], destiné au profilage de l'utilisation des caches CPU.

CACHEGRIND maintient l'état de la hiérarchie de cache, et le fait évoluer au fur et à mesure qu'il reçoit des blocs de bases à instrumenter. Il est capable de gérer des

caches à plusieurs niveaux, unifiés, mais également de Harvard. Le bloc n'est pas altéré par l'instrumentation, mais une analyse est effectuée afin d'altérer l'état du cache simulé. Un bloc de base manipulé par un outil VALGRIND est une suite de traduction en représentation intermédiaire. Un exemple de traduction, issue de la documentation de VEX, associe à l'instruction x86

```
1      addl %eax, %ebx
```

la traduction

```
1  ------ IMark(0x24F275, 7, 0) ------
2      t3 = GET:I32(0)              # get %eax, a 32-bit integer
3      t2 = GET:I32(12)             # get %ebx, a 32-bit integer
4      t1 = Add32(t3,t2)            # addl
5      PUT(0) = t1                  # put %eax
```

Une traduction est composée de deux parties :

- Une instruction `IMark` faisant office d'en-tête. Il s'agit d'une "fausse" instruction dans la mesure où elle ne représente pas une opération, mais indique simplement l'adresse et la taille de l'instruction traduite [1].
- Une suite d'instruction VEX correspondant à la traduction de l'instruction.

`CacheGrind` dépile les instructions les unes après les autres afin de les analyser. Lorsqu'une instruction `IMark` est analysée, le chargement des instructions dans le cache est simulé. Le chargement des données est simulé lorsque l'outil dépile d'une instruction de lecture ou d'écriture en mémoire (exprimée en VEX par les instructions `Ld` et `St`).

Modifications apportées à CacheGrind

Nous avons apporté plusieurs modifications à l'outil CACHEGRIND, afin de pouvoir étudier finement le trafic généré vers le système mémoire.

La première modification permet la capture du trafic généré vers la mémoire. Vu l'abstraction du matériel que nous utilisons (illustrée figure 5.7b), nous considérons qu'un accès vers la mémoire principale a lieu lors des défauts de cache L2. Nous avons modifié le simulateur de cache de Valgrind pour enregistrer ces défauts. Lorsque l'on détecte qu'un accès est émis, nous enregistrons les caractéristiques suivantes :

- L'adresse de destination

1. contrairement aux architectures RISC, les instructions dans les architectures CISC ont des tailles variables

- Le type d'événement à l'origine de l'accès : lecture, écriture ou chargement d'instructions.
- Le nombre d'instructions exécuté entre l'accès courant et l'accès suivant (ou la fin de la capture).

Le nombre d'instructions entre l'accès courant et l'accès suivant permettent de mesurer le progrès de l'application, palliant ainsi l'absence d'information temporelle lors de la simulation. En effet, VALGRIND n'étant pas un simulateur d'architecture, il ne permet pas d'estimer le temps mis pour exécuter le programme.

Afin de relier le trafic enregistré à des mesures réelles, nous avons apporté une modification supplémentaire en ajoutant un système de *points de contrôle*. Un point de contrôle est une instruction spéciale injectée dans le programme, pouvant être vue comme un appel système vers VALGRIND. À chaque point de contrôle est associé un identifiant unique. Lorsque le point de contrôle est exécuté, l'outil enregistre son identifiant, ainsi que la position courante dans le trafic en cours de capture. Les points de contrôles peuvent plus tard être convertis, par exemple, en *points de mesures*, permettant de faire des mesures à l'aide de compteurs de performances. De telles mesures peuvent alors être directement reliées au trafic capturé par notre outil. La fonctionnalité de points de contrôles est implantée en utilisant l'interface `valgrind_monitor_command` de VALGRIND [7].

La dernière modification que nous avons apportée à CACHEGRIND est le suivi des instructions natives en cours d'interprétation. Cette modification nous permet d'enregistrer l'adresse de l'instruction à l'origine d'un accès à la mémoire, nous offrant ainsi la possibilité de localiser le code à l'origine de chaque accès.

5.3.2 Profils haute résolution

La troisième étape du processus de profilage présenté dans la figure 5.8 est une étape de conversion du flux émulé dans un format de profil synthétique. Le format de profil que nous avons adopté est organisé en différentes *couches* représentant chacune un aspect du trafic mémoire. On peut définir une couche comme l'ensemble image d'une fonction associant à un accès n une valeur pour un aspect particulier. Cette organisation est similaire à celle d'un format d'image où chaque pixel est représenté par une succession de couches en décrivant différents aspects (niveau de vert, de rouge, de bleu ou encore transparence). Le trafic tel que capturé peut être représenté à l'aide de trois fonctions (figure 5.10) :

- Une fonction *type* qui donne le type d'un accès.

- Une fonction *adresse* qui donne l'adresse de destination d'un accès.
- Une fonction *distance* qui donne le nombre d'instructions entre un accès et l'accès suivant.

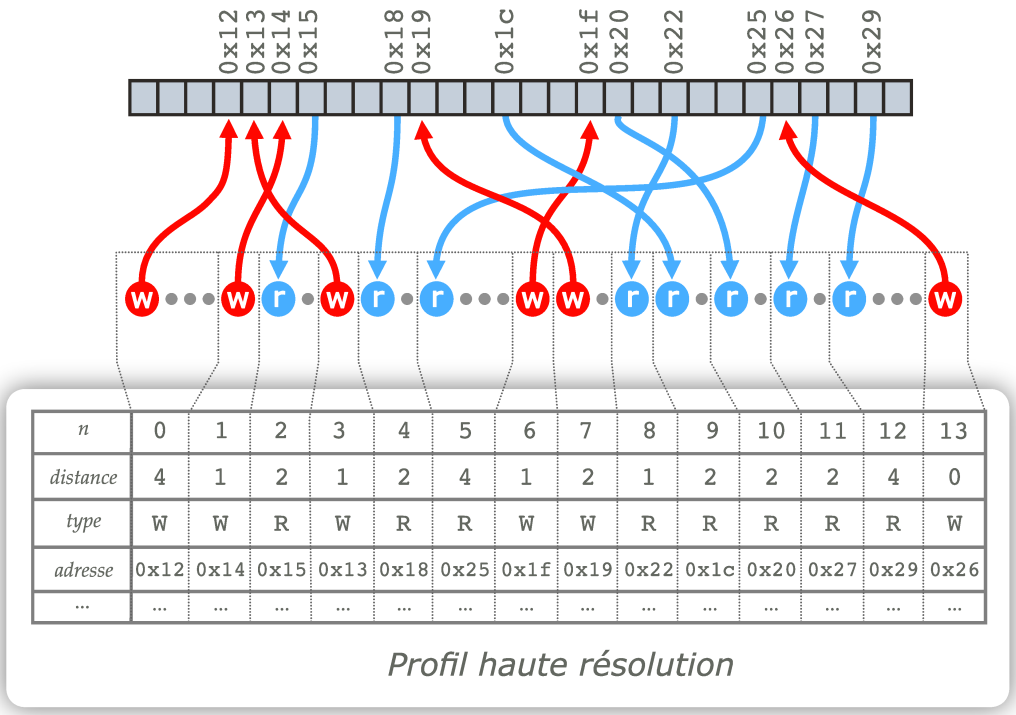


FIGURE 5.10 – Conversion du trafic en profil haute résolution

L'un des principaux avantages de cette représentation est la flexibilité. On peut, en effet, enrichir la représentation du trafic en empilant simplement des couches. De nouvelles couches peuvent être obtenues en étendant le profileur ou en transformant des couches existantes. Par exemple, nous ferons usage d'une opération de somme cumulé. La somme cumulée d'une couche C, à l'accès n est définie par :

$$csum_C(n) = \begin{cases} C(n), & \text{si } n = 0 \\ C(n) + csum_C(n-1), & \text{sinon} \end{cases} \tag{3.IX}$$

Les opérateurs de transformation peuvent également être restreints à certains types de couches. Dans nos travaux, nous restreignons notamment des opérations aux couches

d'adresses. C'est le cas de l'*opérateur de coupe* et de l'*opérateur de sauts*. Comme son nom l'indique, l'opérateur de coupe sert à découper les adresses. Si *addr* désigne une couche d'adresses, $addr[a,b[$ désigne la couche donnant les bits compris dans l'intervalle $[a,b[$ des adresses contenues dans la couche *addr*. Nous utilisons cet opérateur pour découper les adresses de destinations suivant l'interprétation qu'en fait le cache L2.

L'opérateur de sauts est identique à celui présenter dans la section 5.2.3. Il donne le saut effectué entre les adresses consécutives. Soit une couche d'adresses C utilisant des mots permettant de représenter M adresses. La couche de sauts associés à C défini le saut entre l'accès $n-1$ et n par :

$$j_C(n,M) = \begin{cases} C(n), & \text{si } n = 0 \\ (C(n) - C(n-1)) \bmod M, & \text{sinon} \end{cases} \tag{3.X}$$

Les différentes couches à l'œuvre dans nos travaux sont récapitulées dans le tableau 5.1.

TABLE 5.1 – Couches utilisées dans les profils haute résolution

Nom	**Type**[1]	**Définition**
$type(n)$	M	Type de l'accès n
$distance(n)$	M	nombre d'instructions entre l'accès n et $n+1$
$progression(n)$	D	$csum_{distance}(n)$
$adresse(n)$	M	Adresse de destination de l'accès n
$offset(n)$	D	$adresse(n)[0,5]$
$index(n)$	D	$adresse(n)[5,16]$
$tagLSB(n)$	D	$adresse(n)[16,24]$
$tagMSB(n)$	D	$adresse(n)[24,32]$
$dAdresse(n)$	D	$j_{adresse}(n, 2^{32})$
$dOffset(n)$	D	$j_{offset}(n, 32)$
$dIndex(n)$	D	$j_{index}(n, 2048)$
$dTagLSB(n)$	D	$j_{tagLSB}(n, 256)$
$dTagMSB(n)$	D	$j_{tagMSB}(n, 256)$

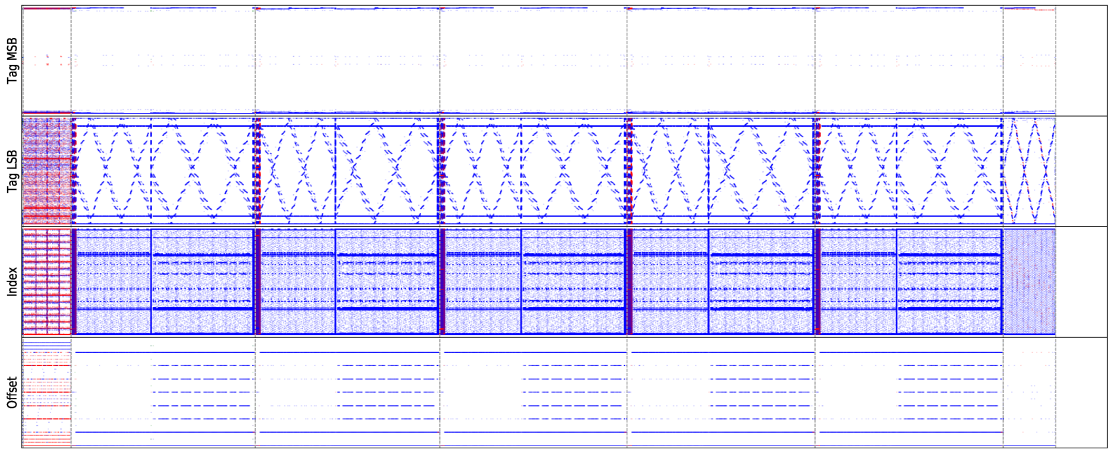
[1] M : mesuré depuis un profil. D : dérivé depuis une autre couche.

Les profils obtenus à l'aide de notre outil peuvent être exploités en les agrégeant ou en les visualisant. L'agrégation consiste à mesurer différentes métriques sur les profils. Notre approche utilise notamment les profils haute résolution pour calculer le taux d'inversion d'accès à la mémoire et l'entropie des différentes couches de sauts entre les adresses. Les points de contrôles permettent notamment d'affiner ces mesures en les faisant sur des portions réduites de profil.

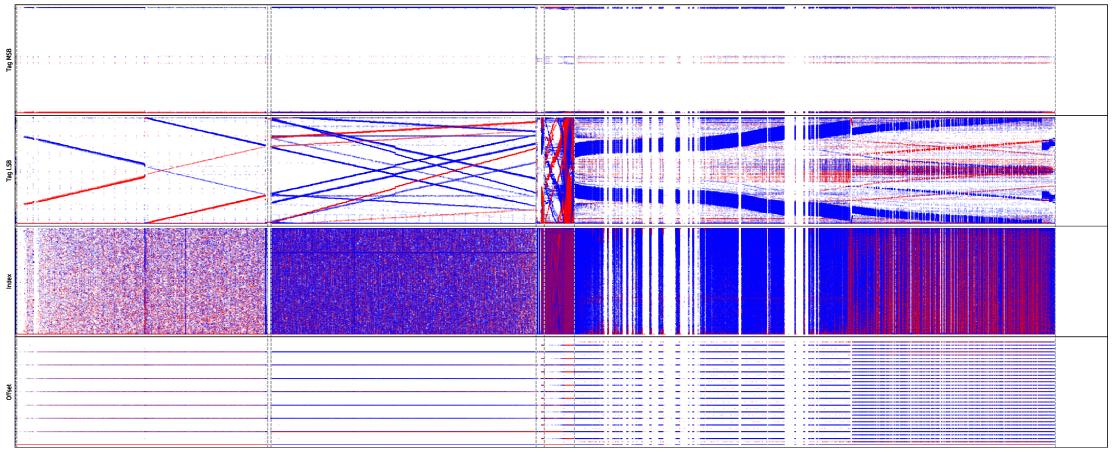
Notre format de profil étant similaire à un format d'image, il peut être visualisé facilement. Des exemples de représentations graphiques pour des applications des suites de benchmarks MIBENCH[38] et PARSEC [21] sont donnés dans la figure 5.11. Notre représentation consiste en quatre graphes superposés représentant chacun la relation entre une couche de saut et la couche de progression de l'application. Les couches de saut ($dOffset$, $dIndex$, $dTagLSB$ et $dTagMSB$) correspondent chacune à une portion des adresses de destination de chaque accès. Le choix de ces couches repose sur l'impact visuel de ces dernières quand elles sont représentées graphiquement. Chaque point représente donc un accès, l'abscisse donne le nombre d'instructions exécutées, et l'ordonnée le saut effectué par rapport à l'accès précédent. Le type de l'accès est représenté par la couleur du point : bleu pour les lectures, rouge pour les écritures, et vert pour les chargements d'instructions.

Sur la figure 5.11 on peut voir des motifs bien distincts d'une application à l'autre. La complexité de ces motifs peut être mesurée à l'aide de l'entropie définie en section 5.2.3. On peut distinguer des comportements de bases dans ces motifs. Dans le cas d'accès séquentiel, les sauts d'une adresse sont constants. Leur représentation forme donc une ligne horizontale(figure 5.12a). À l'inverse, les séquences d'accès aléatoires sont représentées par du bruit(figure 5.12c). Les séquences d'accès représentées par une ligne qui n'est pas horizontale représentent des parcours de la mémoire avec un pas croissant (si la pente de la ligne est positive) ou décroissant (si elle est négative) (figure 5.12d).

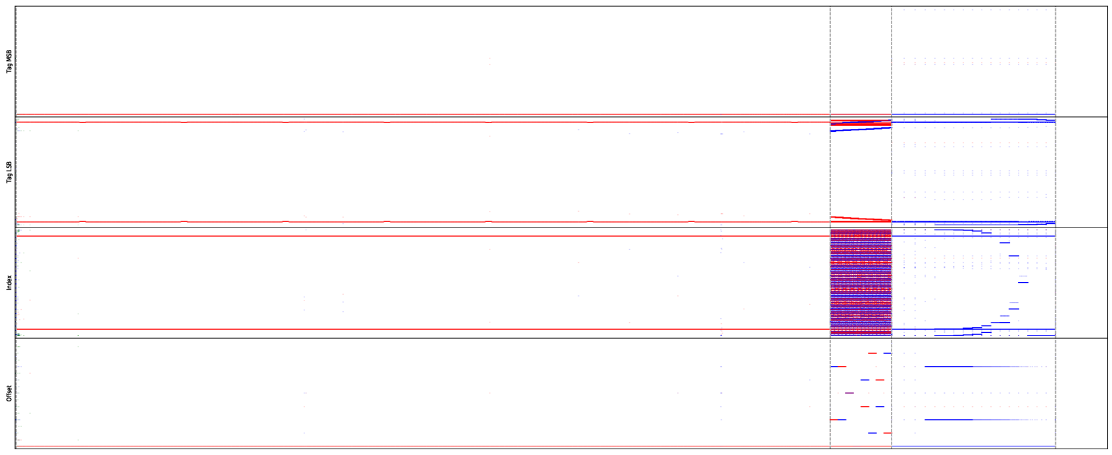
Les motifs représentant les séquences d'accès à la mémoire peuvent se combiner. Cela illustre alors *l'entrelacement* de plusieurs comportements distincts. Un exemple d'entrelacement courant est le parcours en parallèle de plusieurs tableaux 5.12b. La variété des motifs existants, ainsi que la possibilité de les combiner, rend difficile leur catégorisation exhaustive. On peut par contre comparer leur complexité. Par exemple, on peut dire que le motif de la figure 5.12f est plus complexe que celui de la figure 5.12e. À cet effet, l'entropie, présentée dans la section 5.2.3, est un outil utile, car elle permet de quantifier cette complexité.



(a) `fluidanimate-medium`

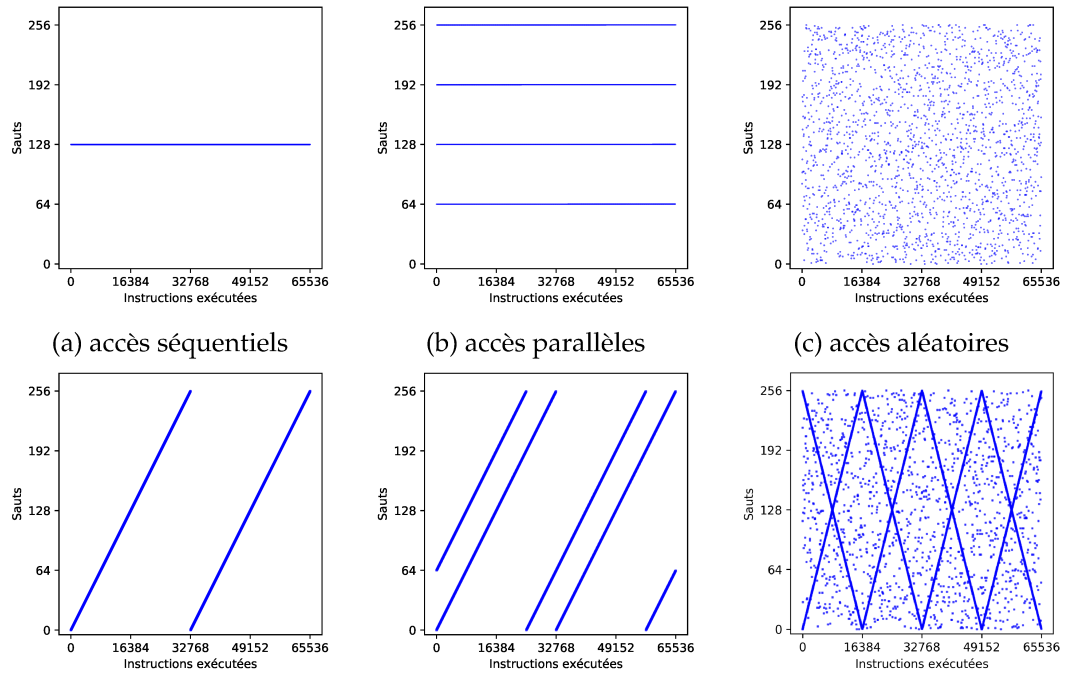


(b) `freqmine-small`



(c) `fft-large`

FIGURE 5.11 – Représentation graphique des séquences d'accès capturés pour des applications des suites de benchmarks MIBENCH et PARSEC. Elle consiste en quatre graphes superposés montrant la relation entre la couche de progression et les différentes couches de sauts telles que définies dans le tableau 5.1. Chaque point de ces graphes correspond donc à un accès, dont le type est encodé par sa couleur : bleu pour les lectures, rouge pour les écritures. Les lignes horizontales en pointillés représentent les passages par des points de contrôles.



(a) accès séquentiels

(b) accès parallèles

(c) accès aléatoires

(d) accès à pas croissants

(e) Entrelacement de deux séquences d'accès à pas croissant

(f) Entrelacement d'accès aléatoire, d'une séquence d'accès à pas croissants, et d'une séquence d'accès à pas décroissants

FIGURE 5.12 – Motifs générés en fonction des types de séquence d'accès

La diversité de motifs peut également se retrouver au sein d'une même application. Cette diversité montre que ces applications ont différentes phases dans leurs exécutions, correspondant à des comportements et donc des sensibilités différentes. La représentation graphique permet donc de valider rapidement le placement de point de contrôle vis-à-vis de ces phases. C'est notamment utile pour des approches de régulation dépendant d'un tel placement [56].

5.3.3 Étude de cas

Nous allons maintenant procéder à une étude de cas sur l'application `bodytrack`. Il s'agit d'une application de tracking vidéo, issue de la suite de benchmarks PARSEC [21], effectuant un traitement sur une série d'images. Nous étudions trois instances de cette application (`small`, `medium` and `large`), variant par la taille des images traitées (et donc de l'empreinte mémoire).

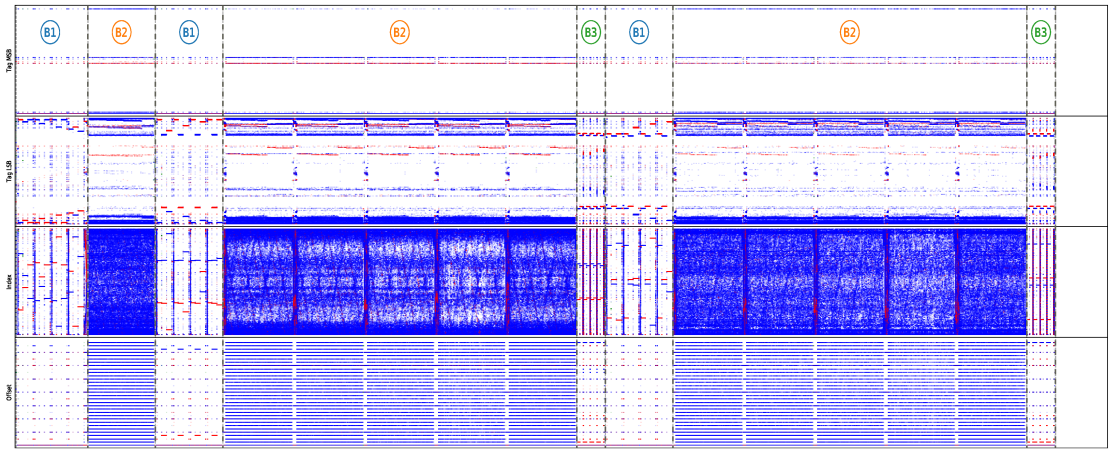


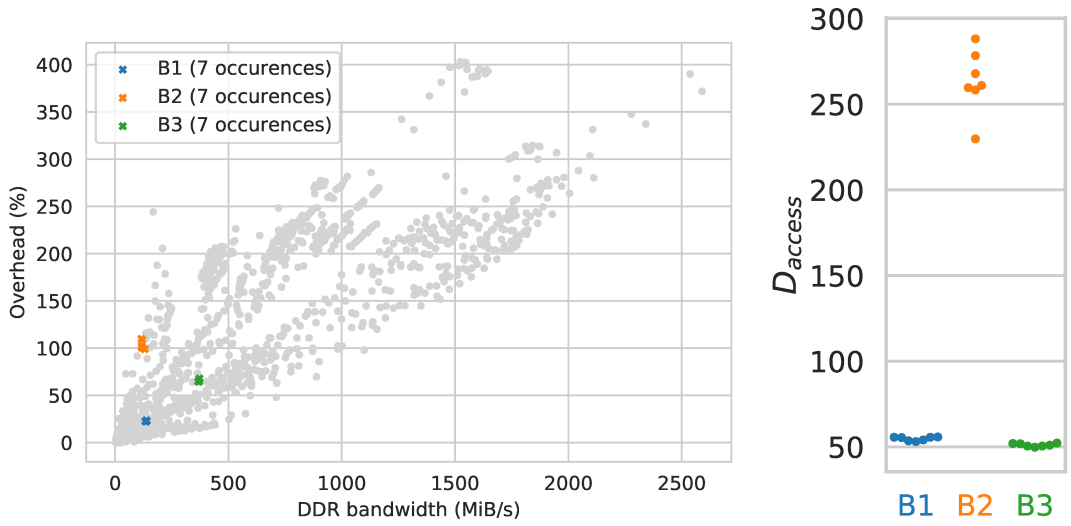
FIGURE 5.13 – Phases de l'application `bodytrack-medium`

L'étude du profil haute résolution des instances de l'application nous permet de distinguer trois comportements distincts, que nous nommerons B_1, B_2, et B_3. L'étude du code et le placement de points de contrôle nous permettent de découper ces instances en un total de 21 phases, chacune d'entre elles correspondant à l'un de ces comportements. La figure 5.11a illustre le découpage de l'instance `medium` avec en annotation le comportement auquel correspond chaque phase. D'après l'examen du code, la phase B_1, correspond au chargement d'une image, la phase B_2 à son traitement, et la phase B_3 à l'écriture du résultat.

La bande passante vers la mémoire principale, le surcoût temporel observé, ainsi que le retard moyen par accès D_{access} de chaque phase est montré dans la figure 5.13. Cette étude est intéressante à trois titres. En premier lieu, on constate que les phases étiquetées avec le même comportement ont des bandes passantes et subissent un surcoût temporel presque identique. Cela montre la pertinence du découpage effectué. En deuxième lieu, on constate à nouveau le manque de précision de la caractérisation par la bande passante. En effet, les phases avec les comportements B_2 et B_3, bien qu'ayant une bande passante presque identique, ont une sensibilité très différentes au problème d'interférences. Le surcoût temporel observé pour B_2 et B_3 étant respectivement de 100% et 20% du temps d'exécution des phases. En troisième et dernier lieu, on constate que si B_1 et B_3 sont visuellements très similaires, les bandes passantes et le surcoût observé pour ces comportements ne le sont pas. Par contre, le retard moyen subi par accès est presque identique [2]. Cela montre l'intérêt de la prise en compte des aspects qualitatifs

2. de manière équivalente, on peut noter sur la figure 5.14 que les points des phases correspondant à

dans la caractérisation du comportement d'accès à la mémoire.



(a) (b)

FIGURE 5.14 – Sensibilité des différentes phases de l'application `bodytrack`

5.4 Conclusions du chapitre

Dans ce chapitre, nous avons abordé la caractérisation du comportement d'accès à la mémoire d'une application. Les critères utilisés dans cette caractérisation devant être pertinents pour l'étude de la sensibilité aux phénomènes d'interférences mémoires.

Nous avons commencé par étudier le lien entre la bande passante (caractérisant l'intensité de l'activité mémoire) et le retard subi à cause des interférences. D'une part, nous avons montré qu'il y effectivement une forte corrélation entre ces deux quantités : une bande passante élevée implique des retards importants. D'autre part, nous avons montré que la plage de retards observés pour des bandes passantes presque identiques était très élevée, y compris pour des bandes passantes faibles. Ainsi, la bande passante utilisée seule se révèle très imprécise pour caractériser la sensibilité aux interférences.

Afin de caractériser plus finement les comportements mémoires, nous avons introduit plusieurs *métriques qualitatives* quantifiant des aspects liés à la nature du trafic. Certaines de ces métriques n'étant pas directement mesurables au moyen de compteurs de performances, nous avons développé un outil de profilage permettant de les mesurer.

ces comportements sont alignés avec l'origine.

À l'aide de cet outil, nous pouvons distinguer des phases caractéristiques dans l'exécution d'applications. Dans une étude de cas, nous avons montré que ces phases pouvaient être affectées très différemment par les interférences.

Dans la section suivante, nous allons juger de la pertinence des différentes métriques définies dans ce chapitre pour l'étude de la sensibilité aux interférences. À cette fin, nous allons évaluer la précision qu'elles permettent d'atteindre lorsqu'elles sont utilisées pour l'inférence du retard causé par les interférences à des applications quelconques.

CHAPITRE 6

Inférer le retard subi

Dans les chapitres précédents, nous avons introduit des microbenchmarks couvrant un large spectre de sensibilité aux interférences et un ensemble de métriques caractérisant leur comportement en isolation. Grâce à ces deux outils, nous disposons d'une grande quantité de données sur la relation entre ce comportement et cette sensibilité. Dans ce chapitre, nous nous penchons sur l'utilisation de ces données pour capturer le comportement de notre plateforme matériel. Le but étant de pouvoir prédire le retard subi par un programme sans avoir à injecter de charges. Pour cela, nous allons nous reposer sur des algorithmes d'apprentissage automatique pour apprendre le comportement du matériel à partir des données dont nous disposons. Par la même occasion, nous allons évaluer quelles métriques sont pertinentes pour caractériser la sensibilité aux interférences.

Ce chapitre comprend deux sections. Dans la première, nous présenterons l'algorithme d'apprentissage que nous utilisons. Dans la seconde, nous procéderons à l'évaluation proprement dite, nous y présenterons d'abord les applications utilisées pour entrainer et valider les résultats de l'algorithme de régression, puis les différents ensembles de caractéristiques évalués, avant de présenter les résultats de l'évaluation.

6.1 Algorithme d'apprentissage

Le but d'un algorithme d'apprentissage est d'approximer une fonction à partir de données. On cherche donc à construire une fonction f dont le but est de prédire la valeur d'une variable y à partir d'un vecteur de caractéristiques X.

$$y = f(X)$$

La nature de la variable y détermine le type de problème que l'on essaie de résoudre. Si y appartient à un ensemble fini et discret, on parle de *classification*. Un exemple de problème classification est de déterminer si une image représente un objet en particulier. Dans le cas où y appartient à un ensemble continu, on parle de *régression*. Par exemple, un modèle climatique prédisant l'évolution de la température moyenne sur terre résout un problème de régression. Nous cherchons à prédire le surcoût temporel subi par une application à cause des interférences. L'ensemble des surcoûts que peut subir une application étant continu , il s'agit donc d'un problème de régression.

Dans ce chapitre, nous étudions la qualité de la prédiction que l'on peut obtenir en fonction de la nature de ce vecteur, c'est-à-dire les différentes caractéristiques de trafic que nous utilisons pour faire la prédiction. L'apprentissage que nous effectuons est *supervisé*. La construction de la fonction d'inférence se fait à l'aide d'un ensemble de données d'entrainement qui consiste en un ensemble de données étiquetées, qui sont des couples (X, y) issus de mesures.

Nous utilisons des forêts d'arbre de décisions pour inférer le retard subi par une application à partir de caractéristiques de leur comportement. Nous avons choisi cet algorithme, car il est à la fois relativement résistant au problème du surapprentissage et simple à utiliser. Le reste de cette section présente brièvement cet algorithme. Nous commencerons par présenter les arbres de décisions qui sont fondamentaux pour la mise en œuvre de celui-ci. Puis nous nous pencherons sur l'algorithme des forêts aléatoires lui-même.

6.1.1 Arbre de décisions

Un arbre de décision est une suite de conditions sur une variable d'entrée X permettant de décider d'une valeur y. Les valeurs de y sont les feuilles de l'arbre, et les conditions sur X ses embranchements. La construction d'un tel arbre à partir d'un ensemble de données est une méthode d'apprentissage permettant de résoudre aussi bien des problèmes de régressions que de classifications. [28].

La construction d'un arbre de décision est décrite dans l'algorithme 2. Elle consiste à découper *récursivement de façon gloutonne* l'ensemble de données d'entrainement en sous-ensemble homogène. L'algorithme prend en entrée un jeu de donnée D et un

compteur de profondeur L. Si les données dans D sont suffisamment homogènes, ou si la hauteur de l'arbre dépasse un seuil préalablement fixé, alors le découpage s'arrête et une feuille est retournée. Sinon une *coupe* est effectuée pour déterminer une condition séparant le jeu de données en deux, et l'algorithme est appliqué récursivement sur les deux sous-ensembles résultant de cette coupe

Algorithm 2 Entrainement d'un arbre de décision

function BUILDTREE((D, L))
  if $L \geq L_{max} \wedge Impurity(D) \leq I_{min}$ **then**
    return $Leaf(D)$
  end if

  $C \leftarrow bestSplit(D)$

  $D_L \leftarrow C(D)$
  $D_R \leftarrow \neg C(D)$

  $T_L \leftarrow BuildTree(D_L, L+1)$
  $T_R \leftarrow BuildTree(D_R, L+1)$

  return $Branch(C, T_L, T_R)$
end function

Les différentes conditions constituant l'arbre de décisions sont déterminées lors des coupes de l'ensemble de données. Pour un ensemble de données D, une coupe optimale C maximise un critère de gain d'information $IG(D, c)$. Avant de définir ce critère, il faut introduire la notion d'*impureté.* L'*impureté* d'un ensemble de données D est une mesure de son hétérogénéité. Plusieurs critères d'impureté existent, mais pour la régression on utilise la *variance* de D. Si N est le nombre d'éléments de l'ensemble d'entrainement et μ la moyenne des y dans D, alors l'impureté de D est

$$Impurity(D) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \mu)^2 \tag{1.I}$$

Un ensemble de données est donc parfaitement homogène lorsque son impureté vaut 0. Vu qu'il s'agit d'un cas que l'on rencontre rarement pour des ensembles de données de plus d'un élément, un ensemble de données est déclaré homogène si son impureté est inférieure ou égale à un seuil I_{min}. Le gain d'information apporté par une coupe c sur des données D est la différence entre l'impureté des données de D et la somme des

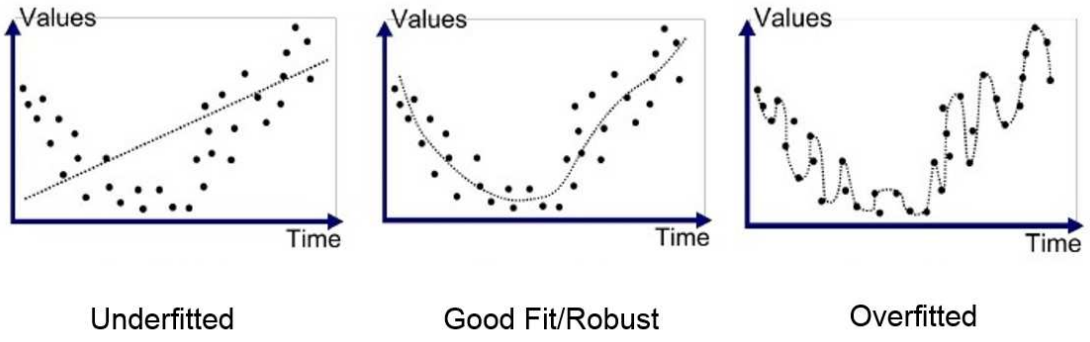


FIGURE 6.1 – Exemples de sousapprentissage, d'apprentissage adapté et de surapprentissage (source [20])

impuretés pondérées des ensembles résultant de cette coupe.

$$IG(D,c) = Impurity(D) - \frac{N_L}{N} impurity(D_L) - \frac{N_R}{N} impurity(D_R) \qquad (1.II)$$

Les arbres de décisions ont pour avantage d'être simples à mettre en œuvre et d'être interprétables par un humain. Mais ils ont pour inconvénient d'être très sensibles au problème de *surapprentissage* (ou *overfitting*). Cela signifie que la fonction apprise s'adapte trop étroitement aux données d'apprentissage, l'empêchant ainsi de se généraliser à d'autres données(figure 6.1). Il s'agit d'une fonction de prédiction avec un biais faible, mais une variance très élevée. On peut réduire cette variance en limitant la hauteur de l'arbre ou à l'aide d'algorithmes randomisés comme les forêts aléatoires. Cette deuxième approche s'avérant particulièrement efficace c'est celle que nous utilisons.

6.1.2 Forets aléatoires

L'algorithme des forêts aléatoires [27] utilise une approche dite d'*ensachage [26] (ou bagging* [1]*)* pour rendre les arbres de décisions plus robustes face au problème de surapprentissage. Il est décrit dans la figure 6.2 et peut se résumer en trois étapes :

1. *Échantillonnage* N sous-ensemble d'entrainements D_i sont tirés aléatoirement *avec remises* à partir de l'ensemble d'entrainements.
2. *Décisions* Pour chaque tirage D_i on entraine un arbre de décision f_i. l'entrainement est similaire à celui décrit dans la section 6.1.1 à une exception près. Soit p le nombre

1. *B*ootstrap *agg*rega*ting*

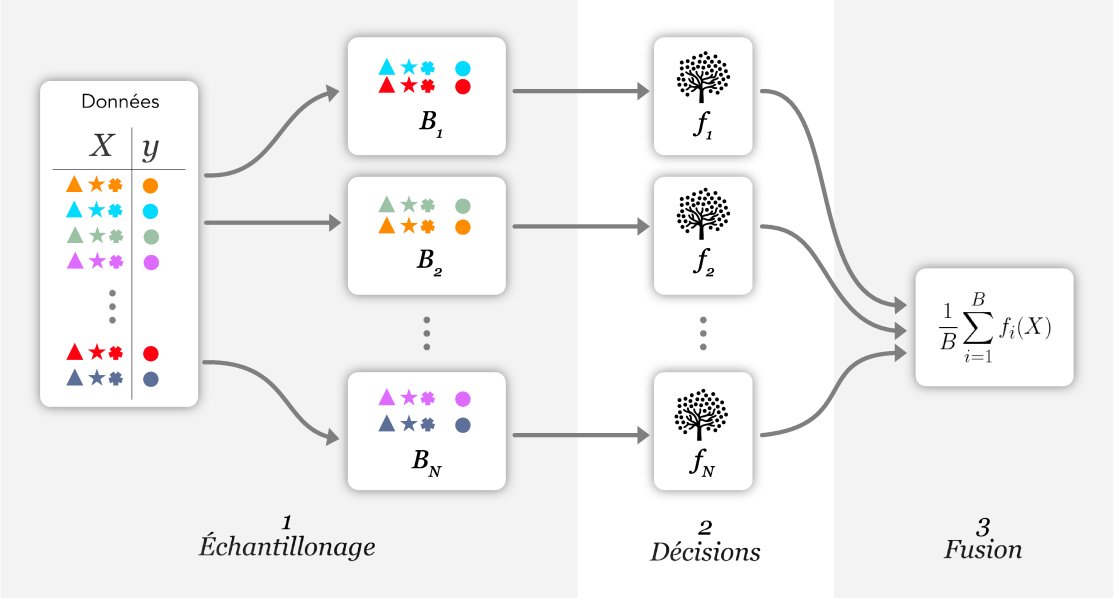


FIGURE 6.2

de composantes de x (le nombre de caractéristiques à partir duquel on veut prédire y). Plutôt que de choisir une coupe optimale parmi toutes les caractéristiques disponibles, on choisit la coupe optimale pour un ensemble de p_{try} caractéristiques tirées aléatoirement.

3. *fusion* pour faire une inférence, on utilise les n estimateurs entrainés durant l'étape précédente et on fusionne leur résultat. Dans le cas d'une procédure de régression, la fusion consiste à prendre la moyenne de la sortie des différents arbres de décisions.

Cet algorithme a l'avantage d'offrir de bons résultats tout en étant simple à utiliser. Il n'ya en effet que deux paramètres à considérer, le nombre d'arbres N que l'on entraine et le nombre p_{try} de caractéristiques que l'on tire pour déterminer une coupe optimale. En règle générale, la valeur de p_{try} utilisée (et celle que nous utilisons [6]) est

$$p_{try} = \lfloor \sqrt{p} \rfloor \tag{1.III}$$

6.2 Évaluation

Nous décrivons maintenant comment nous utilisons l'apprentissage pour inférer le surcoût induit par les interférences. Nous commençons par présenter les différentes applications utilisées pour entrainer le prédicteur et valider sa précision. Nous présentons

ensuite les différents ensembles de caractéristiques que nous utilisons en entrée pour la prédiction. Enfin, nous évaluons quels ensembles de caractéristiques sont les plus adaptés pour notre problème.

6.2.1 Applications

Nous décrivons ici les différentes applications utilisées pour évaluer notre approche. Il y en a deux types : les applications utilisées pour entrainer le modèle de prédiction et celles utilisées pour évaluer la qualité de la prédiction.

Ensemble d'entrainement

Les données de l'ensemble d'entrainement sont issues de nos microbenchmarks. L'ensemble d'entrainement final est en partie constitué de l'ensemble décrit dans la section 4.4.2. Pour les applications du groupe `MemBench`, d'autres instances ont été ajoutées. Ces instances ont des longueurs de rafales 20 et 100. Les répartitions d'accès évaluées sont identiques. Les intensités testées suivent la même règle, mais au lieu d'être comprises entre 0 et 10 000, elles sont comprises entre 0 et 2000.

Ensemble de validation

L'ensemble utilisé pour valider les données est constitué d'applications appartenant aux suites de benchmarks MiBench [38] et PARSEC [21].

Nous utilisons la fonctionnalité de points de contrôle offerte par notre profileur pour découper les applications en différentes phases. Le processus de découpage est actuellement manuel. Il commence par la génération d'un profil haute résolution de l'application, dont l'examen visuel permet d'identifier les différentes phases de l'application. Nous plaçons ensuite des contrôles dans le code des applications. Pour cela, nous nous aidons l'outil `addr2line` pour identifier les lignes de code exécutées dans les phases. Enfin on vérifie visuellement que les points de contrôles sont bien placés.

Les applications évaluées, ainsi que le nombre de phases pour chacune d'elle est donné par le tableau 6.1. Les suites de benchmarks sont généralement constituées de petits programmes n'effectuant qu'une action en particulier. Cela explique que pour beaucoup d'applications, il n'y a qu'une seule phase (comme illustré dans la figure 6.3b). C'est surtout le cas pour les applications issues de la suite MiBench. Néanmoins, certaines applications sont suffisamment complexes pour avoir plusieurs phases (comme illustré dans la figure 6.3a).

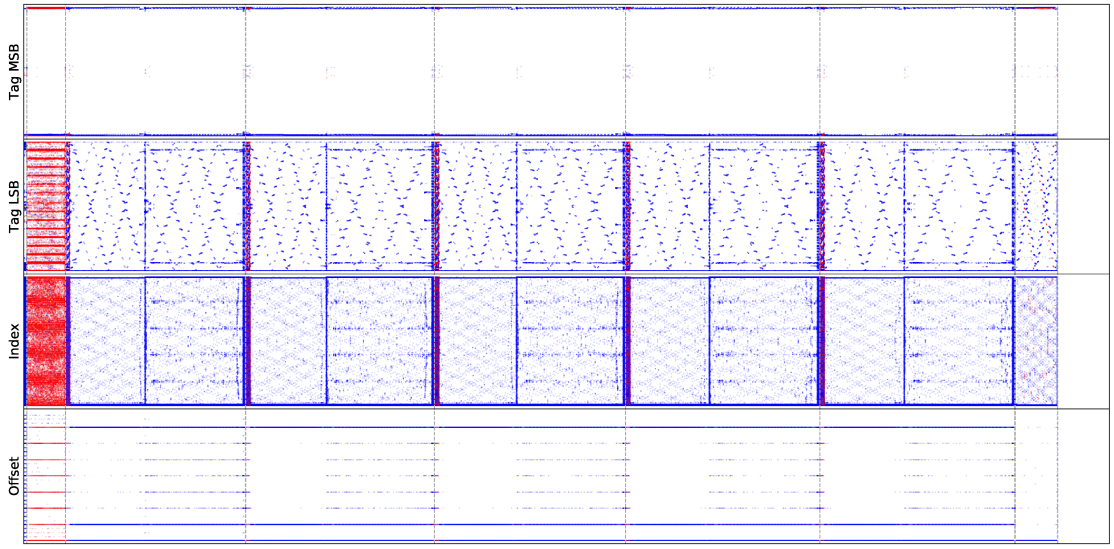
Un des aspects importants qui rentre en compte lorsque l'on identifie les phases est la *granularité* à laquelle on observe le comportement d'accès à la mémoire. Ce critère détermine notamment la taille des phases. Des phases courtes permettent, certes, d'isoler plus de comportement que des phases longues, mais apportent deux inconvénients. D'abord, plus une phase est courte, plus elle impactée par ce qui passé avant qu'elle débute. Ensuite, effectuer des mesures à grain fin peut être difficile. Pour ces raisons, nous avons choisi de limiter la granularité des phases. Ainsi, nous nous limitons à l'étude de phases de plus de 3 millisecondes.

			Nombre de points		
Suite	Benchmark	Description	`small`	`medium`	`large`
PARSEC	blackscholes	Application financière	1	1	1
	`bodydtrack`	Tracking vidéo	3	6	12
	`canneal`	Optimisation	1	1	-
	`fluidanimate`	Simulation de fluide	5	5	5
	`freqmine`	Fouille de donnée	6	6	6
	streamcluster	Clustering	3	-	-
	swaptions	Application financière	1	1	-
MiBench	adpcm-c	Encodage audio	1	-	1
	adpcm-d	Décodage audio	1	-	1
	fft	Transformée de Fourrier rapide	2	3	-
	patricia		1	-	1
	qsort	Tri	-	-	1
	rijndael-dec	Chiffrement	1	-	1
	rijndael-enc	Chiffrement	1	-	1
	sha	Hachage	1	-	1

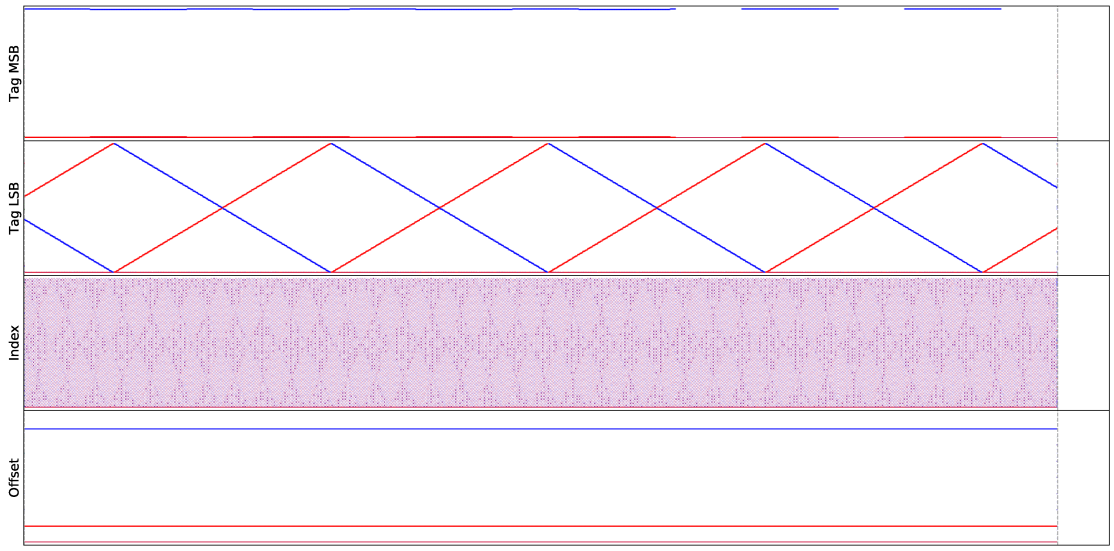
TABLE 6.1 – Applications de l'ensemble de validation

6.2.2 Ensemble de caractéristiques

Notre évaluation porte sur la pertinence de tel ou telle caractéristique de l'activité mémoire en isolation pour prédire le surcoût temporel induit par les interférences sur le système mémoire. Nous considérons les caractéristiques présentées dans le chapitre précédent et résumées dans le tableau 6.2.



(a) `fluidanimate-small.pdf`



(b) `adpcm-d-small.pdf`

FIGURE 6.3 – Exemples de découpage d'applications de l'ensemble de validation

Caractéristique	Description	Source	Page	Section
BW_{iso}^{DDR}	Bande Passante (DRAM)	M	88	5.1
BW_{iso}^{L2}	Bande passante (Cache L2)	C	88	5.1
RW	Taux de lecture / écritures	M	95	5.2.1
DBI	Inversion du bus de données par accès	V	95	5.2.2
$\overline{H}$	Entropie normalisée des accès (bits [0,31])	V	96	5.2.3
H_{offset}	Entropie des accès (bits [0,4])	V	96	5.2.3
H_{index}	Entropie des accès (bits [5, 15])	V	96	5.2.3
H_{tagLSB}	Entropie des accès (bits [16, 23])	V	96	5.2.3
H_{tagMSB}	Entropie des accès (bits [24, 31])	V	96	5.2.3
τ	Impact du temps de service	MC	99	5.2.4

TABLE 6.2 – Caractéristiques évaluées

Afin de limiter le nombre de tests, nous regroupons ces caractéristiques dans différents ensembles, résumés dans le tableau 6.3. Ces ensembles peuvent être divisés en trois catégories :

- Le groupe des *caractéristiques quantitatives* regroupe les ensembles contenant les différentes bandes passantes,
- Le groupe des *métriques qualitatives* regroupe les métriques ayant un rapport avec les types de requêtes effectuées ou encore la complexité des séquences d'accès.
- Le facteur τ (défini en section 5.2.4) est une mesure de l'impact du temps de service des accès DRAM sur la vitesse de progression de l'application. Ce que ce facteur mesure étant différent des autres métriques qualitatives, nous avons choisi de l'isoler dans sa propre catégorie.

Notre évaluation portera donc sur les trois ensembles du groupe quantitatif. Pour chacun de ces ensembles, nous évaluerons le gain apporté par les métriques qualitatives (en ajoutant l'ensemble Q) seules et en conjonction avec τ.

6.2.3 Résultats

L'évaluation porte sur la précision que l'on peut atteindre pour l'inférence de surcoût temporelle en fonction des caractéristiques que l'on prend en entrée. Nos expériences portent sur trois points :

- L'impact de la caractérisation quantitative employée.

Nom	Définition	Groupe
B^{DRAM}	$\{BW_{iso}^{DRAM}\}$	Quantitatif
B^{L2}	$\{BW_{iso}^{L2}\}$	
B	$B^{DRAM} \cup B^{L2}$	
RW	$\{RW, DBI\}$	Qualitatif
H	$\{\overline{H}, H_{offset}, H_{index}, H_{tagLSB}, H_{tagMSB}\}$	
Q	$RW \cup H$	
τ	$\{\tau\}$	τ

TABLE 6.3 – Ensembles de caractéristiques évaluées

- Le gain éventuel apporté par les caractéristiques qualitatives.
- Le gain éventuel apporté par le facteur τ.

Nous mesurons la qualité de la prédiction à l'aide de deux métriques : l'erreur quadratique moyenne (ou MSE pour Mean Squared Error) et l'erreur absolue moyenne (ou MAE pour Mean Absolute Error) de l'ensemble de validation. Si M désigne un ensemble de points à évaluer, alors ces deux métriques sont définies ainsi :

$$MSE(M) = \frac{1}{|M|} \sum_{m \in M} (predicted(m) - observed(m))^2 \tag{2.IV}$$

$$MAE(M) = \frac{1}{|M|} \sum_{m \in M} |predicted(m) - observed(m)| \tag{2.V}$$

Ces deux métriques ont deux rôles différents. La MAE donne une mesure directement interprétable de l'erreur observée pour l'ensemble de validation. La MSE est utilisée pour déterminer l'ampleur des erreurs de prédictions sur l'ensemble de validation. En effet, cette métrique donne un poids plus important aux applications proportionnellement à l'erreur observée. Elle pénalise donc les ensembles de caractéristiques entrainant des erreurs de mesures conséquentes, même si ces dernières ne concernent qu'un nombre réduit de points. La fonction *predicted* donne le plus grand surcoût temporel observé pour une application, celui-ci étant déterminé à l'aide du protocole défini dans la section 4.4.1.

La figure 6.4 illustre le résultat de l'inférence pour les différents ensembles de caractéristiques que nous évaluons. Plus le nuage de points rouge y est resserré plus l'inférence est précise. Notons tout d'abord l'importance du choix de la caractérisation quantitative. Prises seules BW_{iso}^{DRAM} et BW_{iso}^{L2} se révèlent très imprécises, ce qui n'est pas étonnant au

vu de la dispersion observée dans le chapitre précédent. Ce résultat illustre le manque de précision intrinsèque aux approches de régulations basées l'inspection d'une seule ressource. Par contre, un gain substantiel est observable lorsque l'on combine ces deux bandes passantes (respectivement 92,6% et 76,5% de réduction d'erreur quadratique moyenne par rapport à BW^{L2} et BW^{DRAM}).

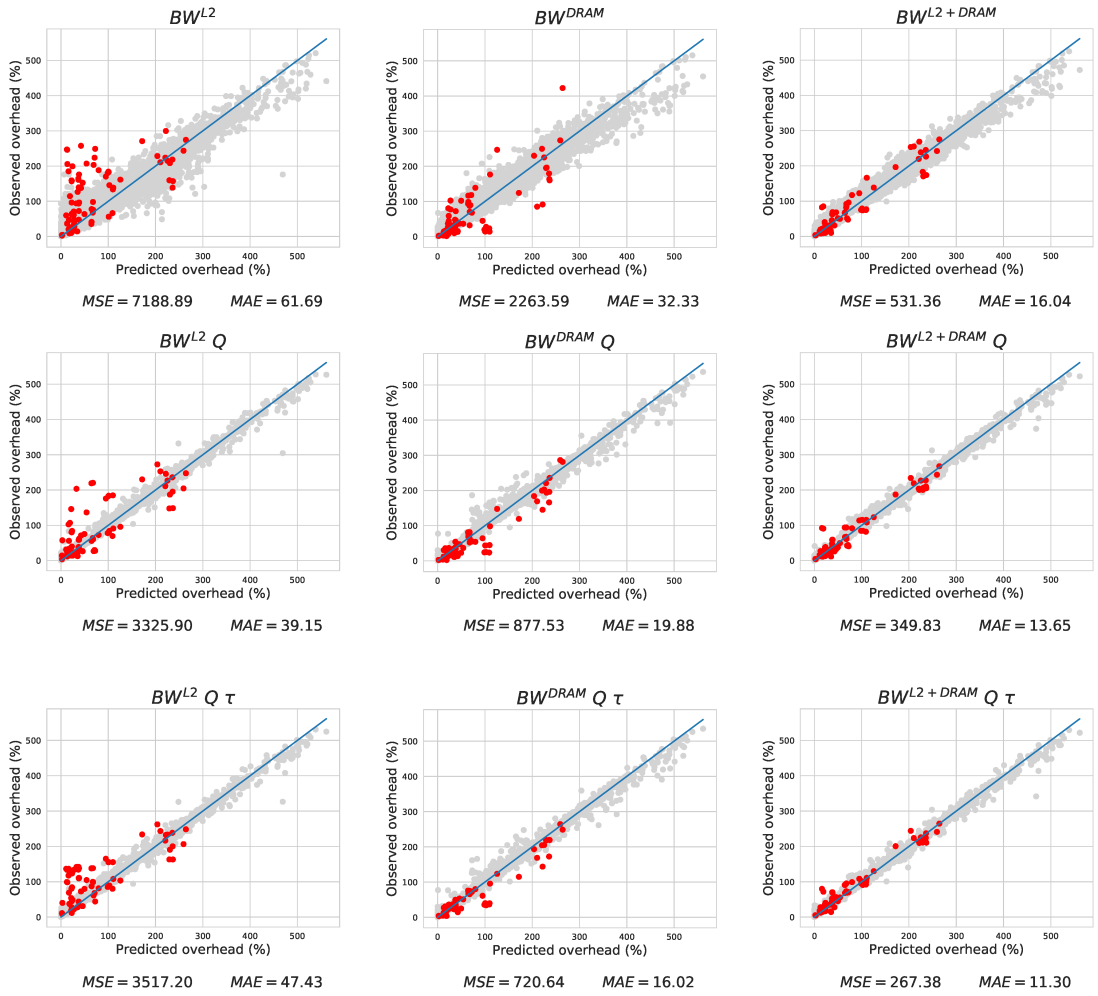


FIGURE 6.4 – Valeurs prédites en fonction des valeurs observées. Les points gris correspondent aux applications de l'ensemble d'entrainement, tandis que les points rouges correspondent aux applications de l'ensemble de validation.

La deuxième observation porte sur l'apport du groupe de caractéristiques qualitatives pour l'inférence de surcoût temporel. L'ajout de métriques qualitatives permet

d'améliorer significativement la qualité de prédiction, et ce quelque soit le groupe de métrique quantitative employée. La réduction d'erreur de prédiction est résumée dans le tableau suivant.

Ensemble quantitatif	BW_{iso}^{L2}	BW_{iso}^{DRAM}	$BW_{iso}^{L2+DRAM}$
Réduction de MSE	55,1%	61,2%	34,2%
Réduction de MAE	36,5%	38,5%	16,2%

TABLE 6.4 – Gain de performance de prédiction apporté par les métriques qualitatives

En moyenne le gain apporté par le facteur τ est relativement modeste, il permet néanmoins d'obtenir la meilleure prédiction. Si on utilise l'ensemble BW_{iso}^{L2} comme caractérisation quantitative, la qualité de prédiction est en moyenne légèrement dégradée. L'examen des erreurs de prédictions maximales (tableau 6.5) révèle néanmoins un gain que ce soit en sur ou en sous-estimation.

6.3 Conclusions

Dans ce chapitre, nous avons utilisé les données issues de nos microbenchmarks pour capturer le comportement du matériel concernant les interférences. Cette étude nous a

Quantitatif	Q	τ	Sous-estimation	Surestimation
			97,47	234,45
BW_{iso}^{L2}	x		88,11	170,69
	x	x	73.72	125.47
			130,64	158,43
BW_{iso}^{DRAM}	x		85,50	27,16
	x	x	78.15	14.76
			62,74	66,39
$BW_{iso}^{L2+DRAM}$	x		31,78	76,21
	x	x	26.16	63.81

TABLE 6.5 – Plus grande sur et sous-estimation observée pour l'ensemble de validation (valeurs absolues)

permis d'évaluer la pertinence de deux de nos outils pour l'apprentissage automatique de ce comportement :

- Nos *microbenchmarks* sont utilisés pour l'entrainement de la fonction de prédiction. On évalue donc si la couverture de comportements qu'ils offrent est suffisante pour permettre à la fonction de prédiction de capturer efficacement le comportement du matériel.
- Nos *différentes métriques* sont utilisées pour la caractérisation du comportement. L'évaluation porte donc également sur la pertinence de l'information qu'elles apportent sur la sensibilité au problème d'interférences.

Les résultats de cette étude montrent plusieurs choses. Tout d'abord, l'utilisation d'une seule caractéristique quantitative ne donne pas de bons résultats. On observe des gains significatifs de précision en prenant en compte les différentes bandes passantes que l'on peut mesurer. Ensuite, on note que l'ajout de métriques qualitatives permet toujours d'améliorer la précision de la prédiction et ceux quelque soit l'ensemble de métriques quantitatives auxquels elles sont associées. De plus, on note une forte réduction dans la sous-estimation du surcoût, ce qui s'avère appréciable dans un contexte temps-réel. L'ajout du facteur τ, permet un gain supplémentaire de précision. Enfin, on peut noter que nos microbenchmarks permettent de capturer assez efficacement le comportement du matériel. En effet, pour l'ensemble de caractéristiques donnant les meilleurs résultats, l'erreur de prédiction absolue observée en moyenne sur l'ensemble de validation est de 11,3%, la plus forte sous-estimation étant d'environ 23%.

Ce chapitre clôt la présentation des différentes contributions de cette thèse. Dans le chapitre suivant, nous allons conclure ce document, et également présenter les différentes perspectives qui s'offrent à nous pour étendre ces travaux.

CHAPITRE 7

Conclusion

Ce chapitre conclut ce document. Nous y récapitulons les contributions apportées par nos travaux. Nous tirons ensuite des conclusions de nos résultats. Nous évoquons, enfin, les différentes perspectives qui s'offrent à nous pour améliorer l'approche que nous avons proposée.

7.1 Résumé des contributions

Au fil de ce document, nous avons produit une approche pour l'inférence du surcoût temporel induit par les interférences à partir d'une caractérisation du comportement en isolation. La figure 7.1 illustre les différents éléments constituant cette approche, ainsi que les chapitres où ils sont définis (ou mis en œuvre). Les chapitres 2 et 3 n'apparaissent pas sur cette figure, car ils exposent respectivement le contexte technique et un état de l'art de la gestion d'interférences dans les systèmes temps-réels.

Le chapitre 4 expose une étude de grande ampleur traitant de l'impact des interférences sur une cible multi-cœur COTS largement utilisée dans l'industrie. Pour conduire cette étude, nous avons développé un ensemble de microbenchmarks permettant de reproduire un grand nombre de comportements d'accès différents, variant aussi bien en *nature* qu'en *intensité*. Cette suite étend des microbenchmarks existants tels que STREAM, en introduisant des paramètres permettant de contrôler leur comportement d'accès à la mémoire. Ces paramètres sont déterminés à l'aide d'une représentation événementielle des accès à la mémoire que nous avons préalablement introduite. Si les résultats de cette première étude confirment que la sensibilité aux interférences est

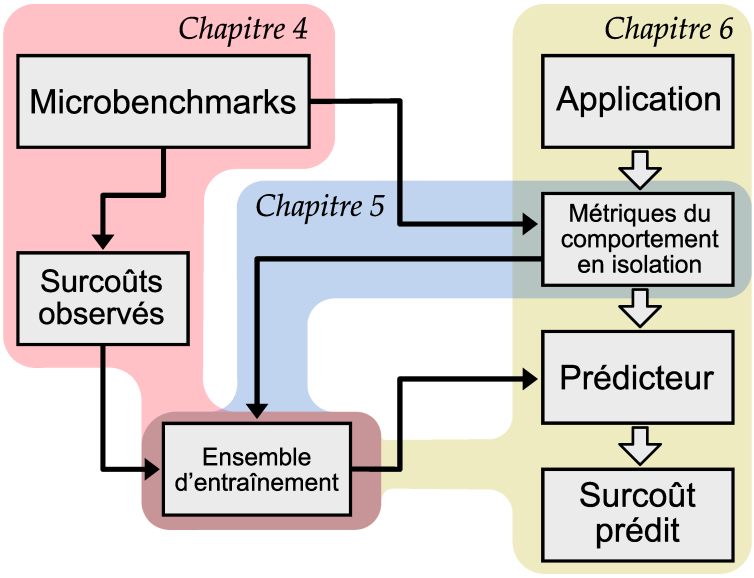


FIGURE 7.1 – Architecture globale du mécanisme d'inférence défini dans nos travaux

fortement corrélée à l'intensité du trafic, ils mettent aussi en évidence une forte influence de la nature des accès. Notons que les microbenchmarks nous ont permis de constituer un ensemble de données nécessaire à la suite de nos travaux reposant sur des approches d'apprentissage automatique, comme le montre la figure 7.1.

Dans le chapitre 5, nous nous sommes penchés sur la caractérisation des comportements applicatifs. Le but étant alors de déterminer un ensemble de métriques, permettant de mesurer *à priori* la sensibilité d'une application à la contention mémoire. Parmi les métriques proposées, nous distinguons les métriques *quantitatives* quantifiant l'intensité du trafic généré par les applications des métriques *qualitatives* distinguant leur nature. Dans cette deuxième étude, nous montrons qu'une caractérisation purement quantitative ne permet pas d'évaluer finement la sensibilité aux interférences, montrant ainsi la nécessité des métriques qualitatives.

Certaines métriques qualitatives que nous avons définies ne sont pas mesurables directement par des moyens classiques. Nous avons résolu ce problème en nous appuyant sur une simulation de la représentation événementielle introduite dans le chapitre 4. Pour ce faire, nous avons développé un prototype de profileur dans le framework d'instrumentation binaire dynamique VALGRIND. En plus de permettre la mesure de métriques qualitatives, ce profileur offre la possibilité de générer des profils en haute résolution de l'activité mémoire, permettant notamment d'identifier des phases caracté-

ristiques. Nous avons ainsi pu utiliser l'outil de profilage pour découper des applications issues des suites MIBENCH et PARSEC en phases homogènes.

Enfin, dans le chapitre 6, nous utilisons les données issues de nos microbenchmarks pour prédire le retard subi par des applications à partir de métriques caractérisant leur comportement en isolation. Cette prédiction repose sur des forêts aléatoires, un algorithme d'apprentissage automatique, entraîné avec des données associant des comportements en isolation à des surcoûts observés. En plus de fournir une estimation du retard, ces prédicteurs nous permettent d'évaluer la pertinence des différentes métriques définies dans le chapitre 5. En comparant la précision des prédictions obtenues en employant tel ou telle métrique, on observe un gain significatif de précision apporté par les métriques qualitatives. L'erreur quadratique moyenne de l'ensemble de tests pouvant être réduite jusqu'à 61,2%.

7.2 Conclusions de la thèse

Comme nous avons pu le voir dans le chapitre 3, le problème des interférences est un obstacle majeur pour l'adoption des processeurs multi-cœurs COTS dans les applications critiques. Ce problème est l'objet d'un grand nombre de publications scientifiques. Les expériences conduites dans le cadre de nos travaux confirment que l'impact de ces interférences ne peut être négligé, dans la mesure où le prendre en compte conduirait à perdre tout le gain apporté par le multi-cœur. Le défi est alors de pouvoir utiliser le matériel en étant suffisamment performant pour tirer parti du parallélisme.

Il est donc nécessaire de considérer individuellement chaque application en caractérisant leur comportement au travers de métriques spécifiques. Les résultats de notre évaluation ont mis en évidence l'importance du choix de ces métriques et nous permettent de tirer les conclusions suivantes :

- Les métriques quantitatives, bien que peu précise lorsqu'elles sont utilisées seules, restent fondamentales pour caractériser la sensibilité des applications. En effet, il y a un lien évident entre la consommation de ressources partagées et la sensibilité aux interférences. Les gains de précisions obtenus en caractérisant la bande passante au travers de plusieurs métriques (bandes passantes vers le cache L2 et vers la DRAM) sont significatifs : on constate une réduction de la MSE de 75% en employant les bandes passantes vers le cache L2 et la DRAM comparé à la bande passante vers la DRAM seule. La conclusion est donc qu'il faut intégrer la bande passante vers

un maximum de composants du système mémoire, y compris si ceux-ci ne sont pas affectés par des interférences spatiales, comme c'est le cas pour le cache L2 de notre plateforme matérielle.

- Les métriques qualitatives permettent d'améliorer la qualité de prédiction, et ce quelque soit la caractérisation quantitative employée. Ces métriques caractérisent les aspects liés à la nature de l'activité mémoire des applications. Ces aspects jouent un rôle sur l'impact d'une dégradation du temps de service des requêtes d'accès à la mémoire, mais aussi sur l'ampleur de cette dégradation. Sur notre plateforme matérielle, le gain de précision apporté par ces métriques peut s'expliquer par la prévalence des interférences temporelles. En effet, les retards induits par ces interférences ont majoritairement pour origine des arbitrages d'accès, qui ne sont pas toujours équitables. Sur certains composants, notamment le contrôleur DRAM (voir section 4.1.5), la nature du trafic peut conditionner la priorité qui lui est donnée. C'est pourquoi il s'avère pertinent de les prendre en compte.
- Nous avons aussi défini, le facteur τ qui mesure l'impact du temps de service moyen des accès DRAM sur la vitesse de progression du programme. Le gain de précision apporté par cette métrique met en exergue l'importance des questions de compositionnalités et de recouvrement dans l'analyse d'interférences. En effet, l'accent est souvent mis sur le retard subi au niveau des différents sous composants matériels, mais la propagation de ce retard par l'interaction de ces composants est peu étudiée. Cela peut s'expliquer par la multitude de sous-systèmes dans le matériel moderne, dont les interactions sont difficiles à modéliser. Néanmoins, il s'agit là d'une piste importante à explorer.
- De façon plus générale, on peut conclure que de ces travaux que les approches théoriques basées sur un comptage des accès mémoire, et donc *in fine* de la bande passante conduisent au mieux à un surdimmensionnement du système. Il est donc nécessaire d'étendre ces approches en y ajoutant un modèle théorique des aspects qualitatifs.

7.3 Perspectives

De nombreuses pistes s'offrent à nous pour améliorer les travaux présentés dans cette thèse. À court terme, nous souhaitons nous pencher sur le placement automatique de point de contrôles, rechercher de nouvelles métriques pour la caractérisation, et étendre notre suite de microbenchmarks. À plus long terme, nous souhaitons prédire le

retard subi par une application en fonction de la charge du système, et développer une approche d'inférence conservative.

Perspectives à court terme

Placement automatique des points de contrôles Actuellement, le placement de points de contrôle dans les applications est effectué manuellement. Nous souhaitons l'automatiser. Nos outils nous offrent la possibilité de ne pas faire reposer uniquement ce découpage sur de l'analyse statique. Des approches basées sur l'analyse de nos profils en haute résolution sont également envisageables. Cela comprend aussi bien des approches utilisées notamment en vision par ordinateurs, que des méthodes plus *ad hoc*. Nous pensons, en particulier, que détecter des changements dans l'évolution de l'entropie des accès est une piste intéressante. L'idée serait alors de définir une fonction décrivant l'évolution de l'entropie durant l'exécution, et de chercher des points d'inflexion dans celle-ci.

Extension de la caractérisation La caractérisation présentée dans nos travaux n'étant pas exhaustive, étendre celle-ci demeure un axe de recherche intéressant. Deux pistes nous intéressent particulièrement : La première est de caractériser les dépendances entre les accès mémoire et le code exécuté. Ces dépendances sont en effet déterminantes dans le comportement des pipelines, car elles sont souvent à l'origine de blocage le temps que les données soient rapatriées. La deuxième piste que nous souhaitons explorer est l'approfondissement de l'étude des aspects liés à la compositionnalité de l'architecture, comme nous avons à le faire avec la définition du facteur τ. Cela peut se traduire par une mesure plus précise de ce facteur, ou bien de l'étendre au cas où l'on peut mesurer plusieurs temps de réponse.

Extension des microbenchmarks En étendant notre ensemble de microbenchmarks, on peut couvrir un plus grand spectre de comportements d'utilisation et donc de sensibilité aux interférences. Nous voulons notamment étudier des aspects largement ignorés dans la littérature : ceux concernant le chargement d'instructions. Ces aspects n'étaient pas forcément pertinents dans les applications que l'on retrouve habituellement dans les suites de benchmarks, celles-ci étant généralement constituées de petits programmes dont l'empreinte mémoire du code est faible. Ce n'est pas toujours le cas des applications industrielles. C'est souvent le cas du code généré, par exemple à partir de langages synchrones comme LUSTRE ou MATLAB SIMULINK. Ces outils, très populaires dans l'industrie des systèmes embarqués critiques, génère des codes combinant à la fois une

mauvaise localité, mais aussi une taille importante (de l'ordre de plusieurs mégaoctets). On peut donc s'attendre à un fort impact de la vitesse des chargements des instructions pour ce type de programmes.

La structure des microbenchmarks que nous avons présentée est volontairement très simple. Nous souhaitons expérimenter des structures plus complexes, reproduisant par exemple des caractéristiques d'applications existantes. Dans ce but, une approche possible serait de spécifier le comportement d'accès par un graphe, à partir duquel le code du microbenchmark serait généré. Cela peut passer par l'utilisation d'un langage dédié.

Perspectives à long terme

Adaptation à différent niveau de charges du système Une autre amélioration possible de nos travaux se situe dans la prise en compte de l'activité possible des autres applications. Les cas que nous considérons dans ce manuscrit sont pessimistes, car nous faisons l'hypothèse de l'ignorance du comportement des autres applications. Concrètement, cette prise en compte peut se traduire par la capacité à adapter les prédictions que nous faisons en fonction d'un niveau de charge du système donné et du type d'accès.

Un obstacle auquel on peut s'attendre pour la mise en œuvre de cette extension est la multiplication du nombre d'expériences requises pour obtenir un ensemble de données d'entraînement. Pour surmonter cet obstacle, nous suggérons l'utilisation de *modèles génératifs* pour réduire le nombre d'expériences. Un tel modèle ayant pour but de générer des données d'entraînements à partir des paramètres des microbenchmarks. Cette idée repose sur le fait que la variation du comportement des microbenchmarks en fonction du paramètre d'intensité est prévisible, comme nous l'avons montré dans les chapitres 4 et 5 de ce document.

Inférence conservative Quelle que soit la précision que l'on peut atteindre, il faut garder à l'esprit que le retard inféré grâce à notre approche dans le chapitre 6 n'en demeure pas moins une estimation du retard observable en pratique. La méthode de prédiction que nous employons n'étant pas conservative, le retard estimé peut être moins grand que le retards réel. Ce dernier ne peut donc être utilisé directement comme une borne dans les systèmes temps-réels durs. Cela ne rend pas ce genre d'estimations inutiles pour autant. Elles peuvent en effet être utilisées pour : découper les applications en phases de sensibilité équivalentes, dimensionner le matériel, ou encore servir de points de comparaison aux résultats d'une analyse WCET. Pour le dimensionnement de systèmes

temps-réels dur, il serait appréciable de disposer d'une approche d'inférence conservative. Nous entendons par là une approche garantissant l'absence de sousestimations si l'ensemble d'entraînement est suffisamment couvrant.

Nous sommes en mesure de faire une telle inférence en réduisant la caractérisation du comportement d'une application à sa bande passante. Le retard prédit pour une bande passante b est alors le plus grand retard observé parmi les bandes passantes inférieures ou égales à b. Cette approche fonctionne, car nous supposons la monotonie de la fonction associant le plus grand retard observable à une bande passante.

Malheureusement, il s'avère difficile de faire de même pour une caractérisation à plusieurs variables. En effet, cette dernière ne fonctionne que si la relation entre le comportement et le retard observé est monotone. Or, la notion de monotonie n'a de sens que si les ensembles de départ et d'arrivée d'une fonction sont ordonnés. Si pour une caractérisation avec une seule métrique quantitative (au sens où nous l'entendons dans le chapitre 5) l'ordre est évident, ce n'est pas le cas pour une caractérisation avec plusieurs variables, à plus forte raison lorsque l'on emploie des métriques qualitatives. Le défi qui se pose est donc de trouver un ordre qui puisse s'appliquer à l'analyse d'interférences.

Publications

Les travaux présentés dans cette thèse on fait l'objet des publications suivantes :

Publications internationales

- Cédric Courtaud, Julien Sopena, Gilles Muller et Daniel Gracia Pérez. *Improving Prediction Accuracy of Memory Interferences for Multicore Platforms.* 2019 40th IEEE International Real-Time Systems Symposium (RTSS'19).
- A. Blin, C. Courtaud, J. Sopena, J. Lawall, G. Muller. *"Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System"*, 28th EUROMICRO Conference on Real-Time Systems (ECRTS'16), Toulouse, France (2016)
- A. Blin, C. Courtaud, J. Sopena, J. Lawall, G. Muller *"Understanding the Memory Consumption of the MiBench Embedded Benchmark"*, Netys, Marakech, Morocco (2016)

Publications nationales

- Cédric Courtaud, Xavier Jean, Madeleine Faugère, Gilles Muller et Julien Sopena. *Représentation spatiale et pseudo-temporelle des comportements mémoire d'une application.* Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS'2017).
- Cédric Courtaud, Julien Sopena, Gilles Muller et Daniel Gracia Pérez. *Caractériser l'impact des interférences mémoires dans les systèmes temps réel et multicoeur à partir des comportements applicatifs.* Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS'2019).

Bibliographie

[1] Autosar. URL : http://www.autosar.org/.

[2] Cachegrind : a cache and branch-prediction profiler. URL : http://valgrind.org/docs/manual/cg-manual.html.

[3] Hiperros sa. URL : https://www.hipperos.com/maestro/.

[4] Memcheck : a memory error detector. URL : http://valgrind.org/docs/manual/mc-manual.html.

[5] PikeOS. URL : http://www.sysgo.com.

[6] Scikit learn randomforestregressor documentation. URL : https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html.

[7] valgrind_monitor_command documentation. URL : http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.valgrind-monitor-commands.

[8] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H Moin, Jan Reineke, Bernhard Schommer, et al. Impact of resource sharing on performance and performance prediction : A survey. In *International Conference on Concurrency Theory*, pages 25–43. Springer, 2013.

[9] Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-aware dynamic memory bandwidth isolation with predictability in cots multicores : An avionics case study. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[10] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator : a predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256. ACM, 2007.

[11] Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software*, page 20. ACM, 2014.

[12] ARM. *SABRE Lite Hardware User Manual*, 1.1 edition, July 2011.

[13] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, c.b edition, November 2012.

[14] ARM. *Cortex-A Series, Programmer s Guide*, version : 3.0 edition, June 2012.

[15] ARM. *Cortex A9 MPCore, Technical Reference Manual*, revision : r4p1 edition, June 2012.

[16] ARM. *Cortex-A9, Technical Reference Manual*, revision : r4p1 edition, June 2012.

[17] ARM. *ARM Generic Interrupt Controller Architecture Specification 2.0*, b.b edition, July 2013.

[18] Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and 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*, pages 300–309. IEEE, 2012.

[19] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa : an open toolbox for adaptive wcet analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

[20] Abnup Bhande. What is underfitting and overfitting in machine learning and how to deal with it. URL : `https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it`

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite : Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[22] Jingyi Bin, Sylvain Girbal, D Gracia Pérez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core cots architectures. In *Embedded Real Time Software and Systems conference*, volume 15, 2014.

[23] David Black-Schaffer, Nikos Nikoleris, Erik Hagersten, and David Eklov. Bandwidth bandit : Quantitative characterization of memory contention. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE Computer Society, 2013.

[24] Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall, and Gilles Muller. Maximizing parallelism without exploding deadlines in a mixed criticality embedded system. In *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, pages 109–119. IEEE, 2016.

[25] Shekhar Borkar. Design challenges of technology scaling. *IEEE micro*, 19(4) :23–29, 1999.

[26] Leo Breiman. Bagging predictors. *Machine learning*, 24(2) :123–140, 1996.

[27] Leo Breiman. Random forests. *Machine learning*, 45(1) :5–32, 2001.

[28] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[29] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. Compiler-directed page coloring for multiprocessors. In *ACM SIGPLAN Notices*, volume 31, pages 244–255. ACM, 1996.

[30] John V DeNuto, Stephen Ewbank, Francis Kleja, Christopher A Lupini, and Robert A Perisho Jr. Lin bus and its potential for use in distributed multiplex applications. *SAE transactions*, pages 135–142, 2001.

[31] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*, 2014.

[32] Christian Ferdinand and Reinhold Heckmann. ait : Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.

[33] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. Assessing the suitability of the ngmp multi-core processor in the space domain. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 175–184. ACM, 2012.

[34] Stuart Fisher. Certifying applications in a multi-core environment : The world's first multi-core certification to sil 4. *SYSGO white paper*, 2013.

[35] Justin Funston, Maxime Lorrillere, Alexandra Fedorova, Baptiste Lepers, David Vengerov, Jean-Pierre Lozi, and Vivien Quema. Placement of virtual containers on NUMA systems : A practical and comprehensive model. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 281–294, Boston, MA, 2018. USENIX Association. URL : `https://www.usenix.org/conference/atc18/presentation/funston`.

[36] Sylvain Girbal, Xavier Jean, Jimmy Le Rhun, Daniel Gracia Pérez, and Marc Gatti. Deterministic platform software for hard real-time systems using multi-core cots. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 8D4–1. IEEE, 2015.

[37] David Griffin, Benjamin Lesage, Iain Bate, Frank Soboczenski, and Robert I Davis. Forecast-based interference : modelling multicore interference from observable factors. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 198–207. ACM, 2017.

[38] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench : A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[39] Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th international conference on real-time networks and systems*, pages 299–308. ACM, 2016.

[40] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis : definition and challenges. *ACM SIGBED Review*, 12(1) :28–36, 2015.

[41] Damien Hardy, Thomas Piquet, and 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*, pages 68–77. IEEE, 2009.

[42] Peter Heise, Iris Gaillardet, Haseeb Rahman, and Vijay Mannur. Avionics full duplex ethernet and the time sensitive networking standard. In *IEEE 802.1 Interim Meeting*, 2015.

[43] John L Hennessy and David A Patterson. *Computer architecture : a quantitative approach*. Elsevier, 2011.

[44] i.MX 6Dual/6Quad Applications Processor Reference Manual. *p.3834-3835*.

[45] i.MX 6Dual/6Quad Applications Processor Reference Manual. *p.3852*.

[46] Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, and Mikael Sjödin. The multi-resource server for predictable execution on multi-core platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2014.

[47] Dan Iorga, Tyler Sorensen, and Alastair F Donaldson. Do your cores play nicely ? a portable framework for multi-core interference tuning and analysis. *arXiv preprint arXiv :1809.05197*, 2018.

[48] Xavier Jean. *Hypervisor control of COTS multicores processors in order to enforce determinism for future avionics equipment*. PhD thesis, PhD Thesis, Telecom ParisTech, 2015.

[49] Stefanos Kaxiras and Alberto Ros. A new perspective for efficient virtual-cache coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 535–546, New York, NY, USA, 2013. ACM. URL : `http://doi.acm.org/10.1145/2485922.2485968`, `doi:10.1145/2485922.2485968`.

[50] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4) :338–359, 1992.

[51] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 145–154. IEEE, 2014.

[52] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and reducing memory interference in cots-based multi-core systems. *Real-Time Systems*, 52(3) :356–395, 2016.

[53] Namhoon Kim, Bryan C Ward, Micaiah Chisholm, James H Anderson, and F Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 53(5) :709–759, 2017.

[54] Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M Petters, and Henrik Theiling. Multicore in real-time systems–temporal isolation challenges due to sha-

red resources. In *Workshop on industry-driven approaches for cost-effective certification of safety-critical, mixed-criticality systems*, 2013.

[55] Yogen Krishnapillai, Zheng Pei Wu, and Rodolfo Pellizzoni. A rank-switching, open-row dram controller for time-predictable systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 27–38. IEEE, 2014.

[56] Angeliki Kritikakou, Claire Pagetti, Olivier Baldellon, Matthieu Roy, and Christine Rochange. Run-time control to increase task parallelism in mixed-critical systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 119–128. IEEE, 2014.

[57] Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and 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*, page 139. ACM, 2014.

[58] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553) :436, 2015.

[59] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared data caches conflicts reduction for wcet computation in multi-core architectures. In *18th International Conference on Real-Time and Network Systems*, page 2283, 2010.

[60] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos : A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3) :56–67, 2007.

[61] John S. Liptay. Structural aspects of the system/360 model 85, ii : The cache. *IBM Systems Journal*, 7(1) :15–21, 1968.

[62] Björn Lisper. Sweet–a tool for wcet flow analysis. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 482–485. Springer, 2014.

[63] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[64] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A survey of timing verification techniques for multi-core real-time systems. 2018.

[65] Rainer Makowitz and Christopher Temple. Flexray-a communication network for automotive control systems. In *2006 IEEE International Workshop on Factory Communication Systems*, pages 207–212. IEEE, 2006.

[66] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013.

[67] PL310 Cache Controller Technical Reference Manual. *p.327-328*.

[68] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up : Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

[69] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[70] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and 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*, MICRO-44, pages 374–385, New York, NY, USA, 2011. ACM. event-place : Porto Alegre, Brazil. URL : `http://doi.acm.org/10.1145/2155620.2155664`, `doi:10.1145/2155620.2155664`.

[71] Nicolas Navet. Controller area network : Cans use within automobiles. *IEEE Potentials*, 17(4) :12–14, 1998.

[72] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, May 2012. `doi:10.1109/EDCC.2012.27`.

[73] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118. IEEE, 2014.

[74] Dominic Oehlert, Selma Saidi, and Heiko Falk. Compiler-based extraction of event arrival functions for real-time systems analysis. In *30th Euromicro Conference on Real-*

Time Systems (ECRTS 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[75] S. A. Panchamukhi and F. Mueller. Providing task isolation via TLB coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13, 2015. doi:10.1109/RTAS.2015.7108391.

[76] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.

[77] Rodolfo Pellizzoni and Heechul Yun. Memory servers for multicore systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.

[78] Quentin Perret, Pascal Maurere, Eric Noulard, Claire Pagetti, Pascal Sainrat, and Benoît Triquet. Predictable composition of memory accesses on manycore processors. 2016.

[79] Paul J Prisaznuk. Integrated modular avionics. In *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference@ m_NAECON 1992*, pages 39–45. IEEE, 1992.

[80] Paul J Prisaznuk. Arinc 653 role in integrated modular avionics (ima). In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1–E. IEEE, 2008.

[81] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4) :34, 2012.

[82] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and 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*, pages 99–108, 2011.

[83] Juan M Rivas, Joël Goossens, Xavier Poczekajlo, and Antonio Paolillo. Implementation of memory centric scheduling for cots multi-core real-time systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[84] Theodore H Romer, Dennis Lee, Brian N Bershad, and J Bradley Chen. Dynamic page mapping policies for cache con ict resolution on standard hardware. In *First Symposium on Operating Systems Design and Implementation*, 1994.

[85] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–222. IEEE, 2011.

[86] Freescale Semiconductor. *CoreLink Level 2 Cache Controller L2C-310, Technical Reference Manual*, revision : r3p3 edition, 2012.

[87] Freescale Semiconductor. *i.MX 6Dual/6Quad Applications Processor Reference Manual*, revision : 1 edition, April 2013.

[88] Freescale Semiconductor. *i.MX 6Dual/6Quad Automotive and Infotainment Applications Processors*, revision : 2 edition, April 2013.

[89] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3) :379–423, 1948.

[90] Timothy Sherwood, Brad Calder, and Joel Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of the 13th international conference on Supercomputing*, pages 155–164. Citeseer, 1999.

[91] Livio Soares, David Tam, and Michael Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE Computer Society, 2008.

[92] DDR3 SDRAM Specification. Jesd79, 2010.

[93] Rapita Systems. Rapitime product's homepage. URL : `https://www.rapitasystems.com/products/rapitime`.

[94] George Taylor, Peter Davies, and Michael Farmwald. The tlb slice-a low-cost high-speed address translation mechanism. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 355–363. IEEE, 1990.

[95] Linus Torvald. Cache coloring. URL : `https://yarchive.net/comp/linux/cache_coloring.html`.

[96] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time*

and Embedded Technology and Applications Symposium (RTAS), pages 1–12. IEEE, 2016.

[97] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243. IEEE, 2007.

[98] John Von Neumann and Goldstine Brucks. Preliminary discussion of the logical design of an electronic computing instrument. 1946.

[99] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Outstanding paper award : Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 2013.

[100] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7) :966–978, 2009.

[101] Maurice V Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, (2) :270–271, 1965.

[102] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3) :37–52, 1987.

[103] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383. IEEE, 2013.

[104] Jun Yan and 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*, pages 80–89. IEEE, 2008.

[105] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6) :681–715, 2012.

[106] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015 :905, 2015.

[107] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc : Dram bank-aware memory allocator for performance isolation on multicore plat-

forms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[108] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and 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*, pages 55–64. IEEE, 2013.

[109] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 455–463. IEEE, 2009.

[110] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

[111] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(5) :1443–1456, 2015.

[112] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(5) :1443–1456, 2016.

[113] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, volume 45, pages 129–142. ACM, 2010.