ANNÉE 2013

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Benjamin LESAGE

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique
ISTIC

Architecture multi-cœurs et temps d'exécution au pire cas

Thèse soutenue à Rennes
le 21 mai 2013

devant le jury composé de :

Sandrine BLAZY / Présidente
Professeur à l'université de Rennes 1

Christine ROCHANGE / Rapporteuse
Professeur à l'université de Toulouse III

Pierre BOULET / Rapporteur
Professeur à l'université de Lille 1

Bernard GOOSSENS / Examinateur
Professeur à l'université de Perpignan

Isabelle PUAUT / Directeur de thèse
Professeur à l'université de Rennes 1

André SEZNEC / Co-directeur de thèse
Directeur de recherche à INRIA Rennes

Outside of a dog, a book is a man's best friend.
Inside of a dog, it's too dark to read.
Groucho Marx

Remerciements

Je tiens à remercier les membres de mon jury de thèse pour avoir jugé les travaux réalisés durant cette thèse, pour leur remarques et leurs questions. En particulier, je remercie Sandrine Blazy, professeur à l'Université de Rennes 1, qui m'a fait l'honneur de présider le dit jury. Je remercie Christine Rochange, Professeur à l'Université de Toulouse III, et Pierre Boulet, Professeur à l'Université de Lille 1, d'avoir accepté la charge de rapporteur du présent document. Je remercie Bernard Goossens, Professeur à l'Université de Perpignan, d'avoir bien voulu juger mon travail.

Je tiens à remercier Isabelle Puaut et André Seznec pour avoir assuré la direction des travaux de cette thèse. Ils m'ont offert la chance de partager avec moi leur vision du monde de la recherche. J'ai apprécié leur patience, leur disponibilité et leurs sages conseils. Leur enseignement sera probablement utilisé à bon escient.

Je veux remercier les membres de mon équipe d'accueil ALF, anciennement CAPS, et de l'équipe ACES que j'ai eu l'occasion de côtoyer durant ma thèse. Ils ont participé à rendre cette thèse plus enrichissante scientifiquement, humainement et culinairement.

Mes remerciements s'adressent aussi à ma famille qui m'a supporté depuis le début, bien avant la thèse. Si ils portent le blâme quant à certaines étrangetés de caractère, il faut porter à leur crédit leur compréhension et leur soutien pendant toutes ces années. Rien de ceci, l'auteur de ce document inclus, ne ce serait fait sans eux.

Je veux remercier mes amis les anciens comme les plus neufs pour m'avoir supporté tout ce temps et à plus forte raison pendant la thèse. Ma reconnaissance va à tous les gens, dans un rayon géographique déraisonnable, qui m'ont soutenu durant la thèse.

Je remercie les auteurs dont j'ai eu la chance de parcourir les contributions musicales, visuelles, ludiques, scientifiques et littéraires durant les années passées. Puisse cette maigre contribution elle aussi perpétrer le savoir que *De chelonian mobile*.

Enfin, je tiens aussi à remercier remercier les lecteurs du présent document, qu'ils en achèvent la lecture, ou qu'ils viennent chercher l'inspiration ou leur nom sur cette page.

Table des matières

Table des matières **i**

Introduction **1**

1 État de l'art sur l'estimation du comportement temporel pire-cas de la hiérarchie mémoire **5**
1.1 Estimation de pire temps d'exécution 6
1.1.1 Mesure du temps d'exécution, méthodes dynamiques 7
1.1.2 Méthodes d'analyse statique 8
1.1.3 Méthodes hybrides pour analyse temporelle 10
1.2 Analyse bas niveau, comportement temporel des caches 11
1.2.1 Architecture et propriétés des caches 11
1.2.2 Analyse statique du comportement temporel au pire cas d'un cache 16
1.2.3 Améliorer la prédictibilité des caches 20
1.3 Discussion 23

2 Comportement pire-cas des caches de données partagés en contexte multi-cœur **25**
2.1 Analyse multi-niveau de caches de données 27
2.1.1 Analyse d'un niveau de cache 28
2.1.2 Classification d'accès au cache (CAC) 32
2.1.3 Estimation de la contribution temporelle au pire cas des caches 34
2.2 Impact des interférences sur les analyses de caches de données . . . 37
2.2.1 Estimation des conflits de cache inter-tâches 37
2.2.2 Prise en compte des conflits durant l'analyse des caches . . . 38
2.2.3 Classification des accès aux données partagées 40
2.3 Réduction des conflits pour les caches partagés, utilisation du bypass 41

2.3.1 Calcul des réutilisations entre accès mémoire 42
2.3.2 Heuristiques pour le bypass 43
2.3.3 Analyse de statique caches de données avec du bypass 44
2.4 Expérimentations 45
2.4.1 Conditions expérimentales 45
2.4.2 Caches de données en environnement uni-cœur 47
2.4.3 Caches de données en environnement multi-cœur 53

3 PRETI : Caches partitionnés pour environnements temps-réel 65
3.1 Fondements de la politique de partitionnement PRETI 67
3.1.1 Politiques d'accès et de mise à jour du cache 67
3.1.2 Implémentation du partitionnement 71
3.1.3 Analyse de caches partitionnés avec PRETI 73
3.2 Expérimentations 74
3.2.1 Conditions expérimentales 74
3.2.2 Impact de PRETI sur l'ordonnançabilité des systèmes 78
3.2.3 Impact de PRETI sur les performances mesurées des tâches . 80
3.3 Étendre le comportement des caches PRETI 87
3.3.1 Restriction des accès aux données partagées 87
3.3.2 Contrôle de la croissance des partitions 89
3.3.3 Extensions à d'autres politiques de remplacement, au delà du LRU 89

Conclusion 95

A Politiques de remplacement basées sur PRETI 99

Glossaire 103

Publications de l'auteur 105

Bibliographie 107

Introduction

Les systèmes temps-réel reposent sur la correction des résultats calculés autant que sur la date de leur production. Les tâches s'ordonnent dans le temps et sont contraintes de terminer avant leur échéance. Ces systèmes sont aussi caractérisés par leur nature critique. Le manquement à une contrainte temporelle peut impliquer pour un système strict, p. ex. de type contrôle aéronautique, des pertes humaines ou financières majeures. Par contraste, on distingue les systèmes temps-réel souples pour lesquels une échéance ratée se résout par une baisse de qualité de service, p. ex. la perte d'images lors du décodage d'une séquence vidéo.

Afin de valider un système temps-réel, il faut pouvoir assurer l'existence d'une organisation de ses tâches dans le temps, un ordonnancement, telle que chaque tâche considérée respecte ses contraintes temporelles dans tous les cas, même le pire. Étant donnée une politique d'ordonnancement, le test d'ordonnançabilité associé exige la connaissance du profil temporel, en particulier le pire temps d'exécution (WCET, *worst case execution time*), des applications impliquées.

L'analyse du comportement temporel d'une tâche pour l'estimation de son WCET est donc une étape pivot des méthodes de validation de systèmes critiques. Cette estimation doit satisfaire aux contraintes de sûreté et de précision. La sûreté est la garantie que le pire temps estimé est supérieur au pire temps effectif. Pour être précis, un pire temps estimé doit être au plus proche du pire temps effectif de l'application.

Remplir ces contraintes de sûreté et de précision suppose la prise en compte de l'environnement matériel sur lequel la tâche analysée s'exécute. Cependant, les architectures généralistes sont orientées vers l'amélioration du temps d'exécution moyen des tâches qu'elles hébergent, temps-réel ou non. Si ces optimisations répondent au besoin croissant de performances des systèmes temps-réel, la complexité matérielle engagée compromet la précision des estimations de WCET et sa méconnaissance porte atteinte à leur sûreté.

Les architectures multi-cœurs ont été introduites pour répondre à de tels be-

soins de performances. Composé de plusieurs cœurs d'exécution, un tel processeur autorise l'exécution en parallèle de différentes tâches. À cette première différence avec les architectures dites uni-cœur s'ajoute la mise en commun de certaines ressources matérielles, comme les antémoires (*cache*), pour réduire la duplication de ressources entre les différents cœurs.

Le comportement d'une tâche, en présence d'une ressource partagée, ne dépend plus uniquement de la tâche elle-même mais aussi de ses interactions avec les tâches concurrentes, par l'intermédiaire de ces ressources qu'elles partagent. Ces interactions peuvent impacter le comportement temporel d'une tâche et doivent soit être prises en compte lors de l'estimation de son WCET, soit être contrôlées afin de garantir la prédictibilité des comportements matériels.

Les conflits inter-tâches, liés au partage de ressource sur une architecture multi-cœur, se distinguent essentiellement en conflits d'accès et en conflits d'état. Les conflits d'accès interviennent quand deux tâches veulent accéder à une même ressource au même moment, par exemple le bus mémoire. L'une d'elles subit un délai du fait de l'arbitrage de l'accès à la ressource. Les conflits d'état apparaissent quand l'état d'une ressource peut être modifié par une tâche de sorte que le comportement temporel d'une autre s'en retrouve impacté.

Pour pallier les conflits d'accès en systèmes temps-réel, de nombreuses politiques d'arbitrage ont été définies [33, 61, 83, 3, 74]. Ces politiques offrent la possibilité de borner le délai d'accès à une ressource. Par exemple dans le cadre d'un canal de communication, [61] offre des garanties pour un débit de transfert minimum. En comparaison, peu de méthodes s'attachent à la considération des conflits d'état dans le cadre considéré [91, 58, 37], et exclusivement pour la hiérarchie mémoire.

La hiérarchie mémoire est une série de caches située entre la mémoire principale et le processeur. Son rôle est de combler le fossé entre un processeur et une mémoire principale relativement plus lente. Chaque niveau de la hiérarchie mémoire contient un sous-ensemble de la mémoire principale, plus rapide d'accès que cette dernière, et interrogé en priorité par le processeur lors d'un accès mémoire. Les différents niveaux de la hiérarchie sont interrogés dans l'ordre, du plus proche au plus lointain. La présence ou l'absence de données en cache résulte en des variations de temps d'exécution, même sur de courtes séquences d'instructions, de l'ordre de plusieurs centaines de cycles.

Dans le cadre des architectures multi-cœurs, certains niveaux de la hiérarchie mémoire peuvent être partagés et voir leur contenu modifié par des tâches concurrentes. Ces niveaux sont donc sujets à des conflits d'état. Les premiers travaux

visant à l'estimation du comportement temporel de tâches en présence de tels caches partagés ciblent la prise en compte [58, 37] ou le contrôle [91] des conflits inter-tâches dans le cadre de caches d'instructions.

Contributions et organisation du document

Ce document s'organise autour de deux principales contributions ayant pour objectif la prise en compte de hiérarchies mémoire riches en environnement multi-cœurs et la prévention des conflits inter-tâches sur les caches partagés. Il s'ouvre sur une brève présentation des méthodes existantes pour l'estimation du comportement temporel au pire cas de tâches, en particulier en présence de caches et d'une hiérarchie mémoire complexe (chapitre 1).

Nous présentons ensuite des méthodes permettant l'intégration des caches de données dans l'estimation de la contribution de hiérarchies mémoire au pire temps d'exécution d'applications critiques (chapitre 2). Elles incluent donc la prise en compte par les méthodes d'analyse de niveaux de caches de données privés [55] puis partagés [56], considérant donc les conflits inter-tâches. Afin d'améliorer la précision des estimations temporelles, nous étudions aussi une méthode de réduction des dits conflits reposant sur le mécanisme de court-circuitage de cache (*bypass*).

La seconde contribution étudiée dans ce document est la politique de remplacement pour caches partagés PRETI (PRETI, *partitionned real-time*) [57] (chapitre 3). Ce mécanisme cible non pas la réduction de l'impact des conflits inter-tâches mais isole, dans des mesures connues statiquement, les tâches des effets de leurs concurrentes en termes de contenu du cache. En sus de cette isolation, un cache PRETI brigue l'amélioration, par rapport aux mécanismes explorés précédemment [91], des performances moyennes de tâches non critiques en système de criticité hybride.

Les analyses classiques, basées sur l'interprétation abstraite, souffrent d'une grande complexité limitant leur passage à l'échelle. Cette complexité est aussi rédhibitoire dans le cadre d'analyses itératives [58] ou durant les premières phases de développement d'un système. Nous avons participé à l'identification de schémas classiques de réutilisation pour les caches d'instructions et à la définition d'analyses de contenu ad hoc. Par exemple des instructions consécutives tendent à accéder au même bloc de cache. Une fois passée la première, le bloc est garanti inséré en cache pour les suivantes. Les analyses proposées [36] permettent l'obtention de résultats d'une précision proche de ceux des méthodes classiques à un coût moindre. Pour plus d'informations sur les analyses développées, le lecteur est renvoyé à la

publication issue de cette collaboration [36].

Chapitre 1

État de l'art sur l'estimation du comportement temporel pire-cas de la hiérarchie mémoire

Un système temps-réel se doit d'être valide non seulement vis-à-vis des résultats et des comportements qu'il produit mais aussi de ses contraintes temporelles. Cette validation repose sur une estimation du comportement temporel des applications. Plus spécifiquement, leur pire temps d'exécution (WCET, *worst case execution time*) est utilisé pour assurer qu'elles satisfont à leurs échéances dans tous les cas, même le pire, et à plus large échelle que le système valide ses contraintes temporelles.

Les WCET calculés se doivent d'offrir deux propriétés fondamentales : la sûreté et la précision. Une estimation sûre du pire temps d'exécution d'une tâche est supérieure à son pire temps effectif. Il s'agit d'une propriété cruciale dans le cadre de systèmes temps-réel stricts, où le dépassement d'une échéance peut conduire à des pertes humaines ou financières. Elle peut être plus lâche pour des systèmes dits souples. La précision de l'estimation implique qu'elle est au plus proche du pire temps effectif de la tâche. Une estimation trop lâche conduit à allouer plus de ressources que nécessaire à un système pour garantir sa validation.

Le temps d'exécution d'une tâche n'est pas constant et varie en fonction de nombreux facteurs tant matériels, les mécanismes architecturaux mis en œuvre au niveau du processeur, que logiciels, le système d'exploitation et les tâches concurrentes, et environnementaux, notamment les données d'entrée du programme qui peuvent affecter le chemin suivi dans l'application.

Durant les deux dernières décennies, nombre de travaux de recherche se sont

concentrés sur l'obtention de ces estimations dans le cadre d'architectures unicœurs complexes comprenant par exemple du parallélisme d'instruction. Cet intérêt pour l'obtention d'estimations sûres et précises est porté à la fois par : le coût du pessimisme et de l'imprécision lors de la validation des systèmes, et le caractère critique des systèmes temps-réel.

La croissance en complexité des architectures matérielles répond aux besoins de performances des systèmes généralistes. Dans une moindre mesure, les systèmes temps-réel ne sont pas étrangers à de tels besoins. Les solutions matérielles présentent l'avantage indéniable d'être transparentes aux utilisateurs des systèmes impliqués. Néanmoins, plus une architecture est complexe, moins les estimations temporelles sur cette architecture sont précises.

L'un de ces facteurs de complexité est la hiérarchie mémoire, une suite d'antémémoires (*cache*) assurant la liaison entre le processeur et la mémoire centrale. Chacune d'entre elles contient une image partielle de la mémoire, plus rapide d'accès que cette dernière. De fait, la présence ou l'absence en cache de données requises par une séquence d'instructions peut résulter en un différence de temps d'exécution de plusieurs centaines de cycles processeur.

Organisation

Ce chapitre, dans un premier temps, présente de façon générale les principales classes de méthodes d'estimation de pire temps d'exécution pour les tâches temps-réel (§ 1.1). Ensuite, nous insistons sur les travaux existants pour la prise en compte du comportement temporel d'un mécanisme matériel particulier, celui de la hiérarchie mémoire (§ 1.2). Enfin, ce chapitre est conclu par une discussion sur l'existant mais aussi les questions et problèmes laissés ouverts par les travaux actuels (§ 1.3).

1.1 Estimation de pire temps d'exécution

Les trois principales familles de méthodes d'estimation du comportement temporel d'une tâche sont présentées dans cette section. Les méthodes dynamiques sont basées sur des mesures répétées du temps d'exécution de la tâche dans différents environnements. Les méthodes statiques reposent sur l'analyse statique formelle du code des tâches. Les méthodes hybrides, quant à elles, combinent analyse statique et mesure.

Il est à noter que dans tous les cas, le pire temps d'exécution de la tâche étudiée doit être borné. Le problème de la terminaison d'un programme étant dans le cas général indécidable [95], des contraintes additionnelles sont requises pour les tâches analysées, telle que l'existence de bornes sur le nombre d'itérations de chaque boucle, contraintes réalistes dans ce domaine [1].

1.1.1 Mesure du temps d'exécution, méthodes dynamiques

En théorie, étant données une architecture hôte et une tâche, son profil temporel exact peut être obtenu en effectuant une mesure pour chacune des valeurs de données d'entrée et toutes les configurations environnementales, bref pour toute combinaison possible de facteurs de variation.

Des méthodes génétiques [101] ont été proposées pour diriger la recherche d'une configuration optimale, maximisant le temps d'exécution de la tâche étudiée. L'algorithme génétique [29] travaille sur une population composée de jeux d'entrées pour une tâche donnée. Cette population subit de façon itérative des évolutions avec l'objectif de converger vers une solution produisant le pire temps d'exécution. Toutefois, un tel algorithme peut se retrouver piégé autour d'un minima local, dans une zone restreinte de l'espace des solutions, quand un minimum global est requis pour satisfaire à la propriété de sûreté de la solution.

Un autre axe de recherche est l'exploration et la mesure du temps d'exécution sur la totalité des chemins possibles de l'application cible [105]. Pour réduire le nombre de cas explorés, des conditions de validité sur l'ensemble des chemins évalués ont été proposées, p. ex. « chaque instruction est exécutée au moins une fois par un des chemins de l'ensemble » Mais des conditions trop précises posent un problème de complexité vis-à-vis du nombre de configurations à explorer.

Si les méthodes dynamiques en reproduisant un environnement d'exécution répondent à la contrainte de précision, le pire cas peut toujours éluder la mesure en cas d'exploration non exhaustive des configurations possibles [102] ; pour des systèmes réalistes, le nombre de configurations à considérer est souvent trop important pour qu'une exploration exhaustive soit applicable [79]. Ces méthodes conviennent pour la validation de systèmes temps-réel souples où un comportement divergent de celui estimé met seulement en jeu la qualité de service. Elles s'avèrent aussi utiles pour une estimation rapide du comportement temporel d'applications. Il est à noter que les mesures peuvent être difficiles à réaliser car requérant le matériel hôte et un mécanisme d'observation ne perturbant pas les résultats obtenus (*probe effect* [28]).

1.1.2 Méthodes d'analyse statique

Pour estimer le pire temps d'exécution d'une application, les méthodes statiques ne reposent sur aucune exécution mais sur l'observation et l'analyse de son code source ou de son exécutable. L'analyse se déroule en deux temps. Une analyse bas niveau calcule le pire temps d'exécution de séquences d'instructions prédéfinies dénommées blocs de base. Basée sur un modèle de la machine hôte, elle prend en compte les effets architecturaux. Une analyse dite de haut niveau compose ensuite ces résultats à partir d'une représentation logique du flot du programme.

Le bloc de base, fréquemment considéré comme unité de l'analyse bas niveau, est la plus grande séquence d'instructions consécutives comportant une unique entrée et une seule sortie. Entrée et sortie sont situées respectivement en début et en fin de séquence [2]. Aucune branche n'entre dans la séquence ou n'en sort en dehors de ses deux extrémités. Toutes les instructions de la séquence sont donc exécutées du début à la fin lorsque le bloc de base est exécuté.

Deux principales représentations de programme [2, 67], et donc de flot entre les blocs de base, sont utilisées comme base des méthodes d'analyses de haut niveau. Un programme est communément modélisé sous forme d'arbre syntaxique ou de graphe de flot de contrôle tel qu'illustré dans l'exemple 1.1.

Méthodes à base d'arbres syntaxiques

Les méthodes à base d'arbres [79, 18] représentent le flot du programme sous la forme d'un arbre de syntaxe abstraite extrait le plus souvent depuis le code source. Les nœuds de cet arbre représentent les différentes structures de contrôle du langage source, notamment boucles et branchements conditionnels. Par extension, les feuilles de l'arbre sont des blocs de base. Ceux-ci sont pondérés par leur pire temps d'exécution obtenu par l'analyse de bas niveau. Le pire temps d'une application est calculé lors d'un parcours de bas en haut de l'arbre correspondant. Pour obtenir le poids d'un nœud, une règle de combinaison dépendante de son type est utilisée. Dans l'exemple 1.1, le poids de la structure conditionnelle IF est le poids de la plus lourde de ses branches, WHILE ou BB_4, sommé à celui de l'évaluation de sa condition BB_1.

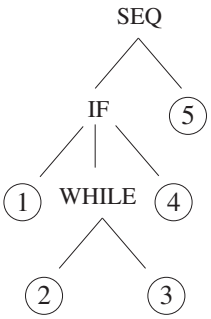
Les méthodes à base d'arbres ont l'avantage de la simplicité et de la rapidité de calcul. Leur proximité de la structure du programme permet l'identification des structures sensibles et coûteuses. Cette même proximité implique la restriction de la représentation à des programmes correctement formés [82], pour lesquels une relation hiérarchique entre les blocs de base est définie. Ceci est un inconvénient

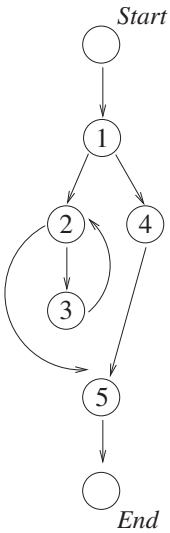
---

EXEMPLE 1.1 – Représentation de programmes

Un simple programme est ici présenté conjointement à sa représentation sous la forme d'un arbre syntaxique et d'un graphe de flot de contrôle. Ce code est composé d'une boucle imbriquée dans une conditionnelle, elle-même suivi d'une séquence d'instructions. Cinq blocs de base ont été identifiés correspondant respectivement : 1 à l'évaluation de la condition, 2 à la tête de la boucle, 3 au corps de la boucle, 4 à la branche ELSE de la conditionnelle, et 5 à des instructions consécutives à la conditionnelle.

IF BB_1 THEN
  WHILE BB_2 DO
    BB_3
  DONE
ELSE
  BB_4
FI
BB_5





L'arbre illustre la structure par trois types de nœuds ; SEQ, IF et WHILE correspondent respectivement à une séquence de blocs de bases, une instruction conditionnelle et une boucle. Le fils gauche des nœuds de type IF et WHILE représente l'évaluation de leur condition.
Dans le cadre de la représentation sous forme de flot de contrôle, la source et le puits du flot sont aussi représentés, respectivement sous la dénomination *Start* et *End*.

---

face aux compilateurs optimisants qui peuvent altérer la structure du programme.

Exploration d'un graphe de flot de contrôle

Un programme peut aussi être représenté sous la forme d'un graphe de flot de contrôle (CFG, *control flow graph*). Il s'agit d'un graphe orienté où les nœuds représentent les blocs de base de l'application. Les arêtes du graphe lient la sortie de chaque bloc de base à l'entrée de ses successeurs pour ainsi modéliser les chemins possibles entre blocs de base. Le pire chemin du point de vue temps d'exécution peut être déterminé par une énumération de l'ensemble des chemins possibles. Une telle énumération explicite est toutefois trop coûteuse dans le cas général.

En lieu et place, les méthodes utilisées reposent sur une énumération implicite des chemins [59] (IPET, *implicit path enumeration technique*). Le calcul du pire chemin est obtenu par résolution d'un système de contraintes en programmation linéaire par nombres entiers [85] (ILP, *integer linear programming*) capturant les relations entre blocs de base consécutifs. La résolution de ce système vise à maximiser une fonction objectif représentant le temps d'exécution du programme.

La maximisation d'une fonction dans un contexte ILP est un problème NP-complet dans le cas général [85]. Toutefois, sur les contraintes générées à partir d'un graphe de flot de contrôle, il est prouvé que le problème se dégrade en problème de flot maximum avec un temps de résolution polynomial [59]. De plus, des contraintes de flots compliquées peuvent être modélisées comme la dépendance entre deux chemins [23]. Toutefois, des contraintes non maîtrisées peuvent mettre en jeu la propriété de résolution en temps polynomial.

1.1.3 Méthodes hybrides pour analyse temporelle

Les méthodes hybrides [17, 12] se situent à mi-chemin entre les mesures des méthodes dynamiques et l'analyse formelle des méthodes statiques. Le temps d'exécution d'une application est mesuré dans différentes configurations environnementales, de façon similaire aux méthodes dynamiques. De ces profils est extrait le temps d'exécution de chaque bloc de base ou de portions plus importantes de la tâche. Ces résultats sont ensuite combinés en utilisant une méthode type haut niveau, p. ex. à base d'arbre syntaxique. Cette approche mitige la perte de précision due aux analyses bas niveau, remplacées par des mesures, mais surtout évite le besoin d'un modélisation complexe de l'architecture hôte. La sûreté de l'estimation résultante reste difficile à assurer.

Les approches probabilistes [11, 20] s'éloignent de cet objectif de sûreté. Elles visent notamment à estimer la répartition des temps d'exécution possibles, répartition d'intérêt pour les systèmes temps-réel souples. Ces estimations reposent sur la combinaison analytique de résultats obtenus lors de mesures répétées du comportement de l'application cible.

1.2 Analyse bas niveau, comportement temporel des caches

Pour estimer le pire temps d'exécution d'une tâche temps-réel, les méthodes statiques requièrent une prise en compte du matériel sous-jacent. Cette prise en compte a lieu dans le cadre d'une analyse dite de *bas niveau*. L'objectif est d'estimer la latence subie lors de l'exécution d'un bloc de base à cause d'un mécanisme architectural. La hiérarchie mémoire est l'un de ces mécanismes. À cause des caches, un même bloc de base peut subir des latences différentes à l'exécution selon le chemin suivi par l'application pour l'atteindre.

Cette section se focalise sur les caches et la prise en compte de la hiérarchie mémoire dans le cadre de l'estimation du pire temps d'exécution d'une tâche. Les fondamentaux du mécanisme sont tout d'abord introduits (§ 1.2.1) afin de mettre en évidence ses propriétés comportementales. Les principales méthodes de prise en compte d'un cache simple et leurs extensions à des hiérarchies et systèmes plus complexes sont présentées (§ 1.2.2). Divers travaux connexes basés sur des mécanismes logiciels ou architecturaux facilitant le travail d'analyse sont finalement mentionnés (§ 1.2.3). Les travaux s'attachant à la prise en compte de l'impact d'autres éléments architecturaux que la hiérarchie mémoire, comme les prédicteurs de branchement [14] ou le pipeline [22] ne sont pas étudiés ici.

1.2.1 Architecture et propriétés des caches

Le but premier d'un cache [88] est de combler la latence entre une mémoire principale lente et des processeurs toujours plus rapides. Un cache contient un sous-ensemble rapide d'accès de la mémoire principale et interrogé en priorité par le processeur quand ce dernier requiert une donnée de la mémoire. Si l'information recherchée est présente en cache, elle peut être servie immédiatement à une latence réduite par rapport à un accès en mémoire principale ; il s'agit alors d'un *succès de cache*. Dans le scénario inverse, dénommé un *défaut de cache*, l'information est

absente du cache et la requête du processeur est relayée à la mémoire principale.

Dans l'éventualité où l'accès à une donnée provoque un défaut de cache, le bloc mémoire complet où se trouve cette dernière est inséré dans le cache fautif lors de la remontée du résultat de la requête. Cette insertion sert les propriétés de localité temporelle et spatiale des programmes. La localité temporelle indique qu'une donnée accédée à l'instant T a une forte probabilité d'être accédée de nouveau à un instant $T + \epsilon$. De façon similaire, la localité spatiale indique que si une donnée A est accédée à un instant donné, les données voisines de A ont de fortes probabilités d'être accédées dans un futur proche.

Organisés en hiérarchie, les caches sont interrogés par le processeur du plus proche au plus éloigné de ce dernier jusqu'à obtention de la donnée recherchée. En cas de défaut, la requête du processeur est donc relayée aux niveaux inférieurs de la hiérarchie mémoire jusqu'à éventuellement atteindre, et être satisfaite par, la mémoire principale.

Les caches peuvent être classifiés respectivement en caches d'instructions, de données ou en caches unifiés selon qu'ils rapatrient de la mémoire :

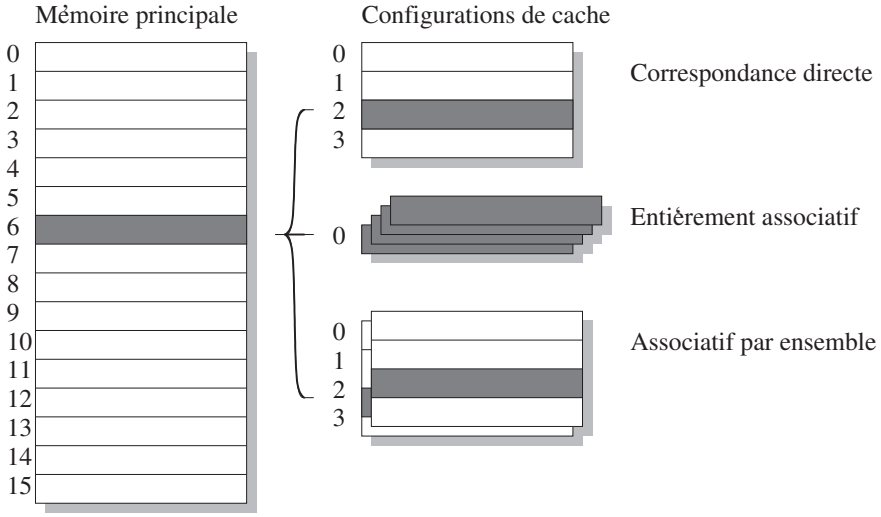
- les instructions exécutées par les tâches sur le processeur ;
- les données sur lesquelles travaillent les programmes, accédées à la demande de certaines instructions ;
- ou bien la combinaison des deux.

Organisation du contenu d'un cache

Comme mentionné antérieurement, les caches sont divisés sous la forme de blocs de taille fixe qui constituent l'unité de transfert depuis la mémoire. De fait, les derniers bits d'une adresse en mémoire indiquent sa position au sein de son bloc de cache, son *offset*. Les bits suivants de l'adresse d'un bloc identifient les lignes de cache qui sont des positions valides pour ce bloc, positions regroupées dans un ensemble de cache ou *set*. Pour un cache donné, la taille de ses ensembles est fixe et dépend de l'associativité du cache. Un cache dont l'associativité est de 1 est dit *direct-mapped*, à correspondance directe ; un bloc ne peut être stocké que dans une ligne particulière. À l'inverse, si l'associativité est égale au nombre de lignes du cache, on parle alors de cache entièrement associatif ; un bloc peut être stocké dans n'importe quelle ligne du cache. Ces différents scénarios sont présentés dans l'exemple 1.2.

Exemple 1.2 – Structure d'un cache

Nous illustrons ici la structure d'un cache et le positionnement d'un bloc mémoire dans le cache en fonction de son associativité. De haut en bas sont illustrées les structures d'un cache à correspondance directe de 4 blocs, un cache entièrement associatif de la même taille, et un cache associatif par ensemble, de 4 ensembles et 2 voies.



L'ensemble des positions valides pour le bloc mémoire numéro 6 est aussi mis en avant pour chacune des configurations présentées. Pour le cache à correspondance directe, seule la ligne 2 peut contenir le bloc 6, ainsi que les blocs mémoire 2, 10 et 14. Deux lignes peuvent recevoir le bloc 6 dans le cache associatif par ensemble, conformément à son associativité.

Classification des défauts de cache

Cette organisation de l'espace du cache conduit à différents types de défauts de cache [41] :

- Les défauts dits *compulsifs* correspondent au premier accès à un bloc de cache ; un bloc qui n'a jamais été demandé auprès d'un cache ne peut y avoir été inséré.
- Les défauts de *capacité* sont dus à la taille limitée du cache ; la donnée recherchée est entrée en cache mais a été évincée suite à un nombre d'accès à des blocs différents suffisant pour remplir la totalité des lignes du cache.

– Enfin, les défauts de *conflit* tiennent à la division par ensembles du cache ; comme pour les défauts de capacité, la donnée est entrée en cache mais a été évincée à cause d'un nombre d'accès assez grand à d'autres blocs stockés dans le même ensemble de cache. Ce nombre d'accès doit toutefois être inférieur au nombre total de lignes du cache ou le défaut est un défaut de capacité.

Les caches entièrement associatifs, ne comprenant qu'un unique ensemble, ne sont par construction pas soumis aux défauts de conflit.

Politique de remplacement

Un bloc mémoire peut être inséré dans n'importe laquelle des lignes de son ensemble. Exception faite des caches à correspondance directe, lors d'une insertion consécutive à un défaut la *politique de remplacement* doit sélectionner la ligne de destination parmi ces différentes candidates. Cela provoque potentiellement l'éviction du bloc contenu auparavant dans cette ligne. Une politique idéale [8] minimise le nombre de remplacements en sélectionnant pour l'éviction le bloc le moins utile dans l'avenir. Toutefois, un tel oracle n'est pas réalisable et des heuristiques sont requises.

Les politiques de remplacement reposent communément sur la création d'une organisation logique entre les différents blocs de chaque ensemble. Cette organisation intervient dans les trois politiques distinctes qui composent la politique de remplacement d'un cache :

– la politique d'*insertion* a trait au statut d'un bloc nouvellement inséré en cache ;

– la politique de *promotion* cible les succès et la position logique du bloc accédé ;

– enfin, la politique d'*éviction* assure lors d'un défaut le choix du bloc évincé et donc de la ligne de destination du bloc en défaut.

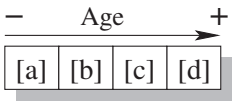
L'exemple 1.3 présente les principes et le fonctionnement d'une de ces politiques de remplacement, la politique LRU (*least recently used*).

Politique d'écriture

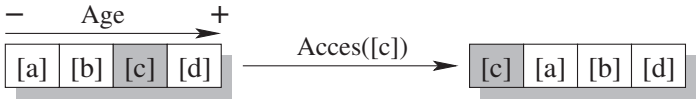
À la demande de certaines instructions, une écriture peut être effectuée en mémoire. Différentes politiques sont mises en œuvre par les caches en réaction à ces modifications de données et régissent la propagation d'une écriture ainsi que son impact sur le contenu du cache en cas de défaut :

Exemple 1.3 – Politique de remplacement LRU

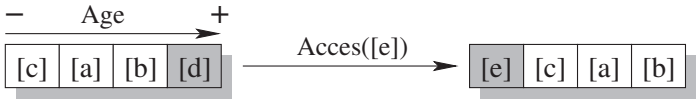
La politique de remplacement LRU ordonne logiquement les blocs dans chaque ensemble selon leur âge du plus récemment utilisé (MRU, *most recently used*) au moins récemment utilisé (LRU). Considérons un ensemble où le le bloc $[a]$ est le plus récent, à gauche, et le bloc $[d]$ est le plus ancien, à droite :



Insertion et promotion propulsent le bloc cible en position MRU, la plus récente. Si le bloc $[c]$ est accédé, un succès de cache dans l'ensemble considéré, il est promu en MRU :



En cas de défaut, c'est le bloc le plus ancien, LRU, qui est évincé. Ainsi, dans l'exemple précédent, si l'on accède au bloc $[e]$ non présent en cache, le bloc $[d]$ doit être évincé afin de libérer une ligne :



- *write-through* implique la propagation de la modification dans la totalité de la hiérarchie mémoire ; tous les niveaux sont modifiés pour prendre en compte la nouvelle valeur.
- *write-back*, seul le niveau courant est mis à jour et le bloc concerné est identifié comme modifié *(dirty)*. La mise à jour est propagée au niveau suivant sur éviction du bloc *dirty*.
- *write-no-allocate* prévient l'insertion du bloc provoquant un défaut lors d'une écriture.
- *write-allocate* correspond au comportement classique sur défaut ; un bloc accédé en écriture est inséré en cache.

Politiques de gestion de la hiérarchie

Des politiques globales peuvent régir et contraindre le contenu d'un cache vis-à-vis du contenu du ou des niveaux suivants de la hiérarchie mémoire. En particulier peuvent être assurés l'*inclusion* du contenu d'un niveau dans le niveau suivant ou l'*exclusion* des contenus de niveaux de caches différents. Si aucune de ces politiques n'est mise en œuvre on parle alors d'une hiérarchie *non-inclusive* ou *mostly-inclusive*.

1.2.2 Analyse statique du comportement temporel au pire cas d'un cache

Succès et défauts, sources de variation de la latence d'accès à un cache, dépendent de l'historique des requêtes servies par ce dernier. Ce comportement dynamique des caches, si il favorise le temps d'exécution moyen des applications, pose un problème de prédictibilité dans le cadre des systèmes temps-réel et de l'estimation de pire temps d'exécution. La différence de temps d'exécution entre un succès et un défaut peut être importante [66] et impacter fortement le pire temps d'exécution même des plus petits blocs de base, à plus forte raison en cas de défauts de cache.

Il est donc primordial de pouvoir estimer le comportement au pire cas des instructions vis-à-vis des caches. Une solution naïve à ce problème consiste à calculer en tout point du programme l'ensemble des états de cache possibles à l'exécution [72]. Si cette méthode assure précision et sûreté, le nombre d'états à maintenir et considérer tend à exploser [96] ; sa complexité dans le cas général est rédhibitoire.

Les premières méthodes [69, 26] pour l'estimation du contenu de caches reposent donc sur l'utilisation d'une représentation *abstraite* des états de cache possibles à l'exécution. Sur les bases d'une analyse de type *dataflow* [96], un état d'entrée est calculé pour chaque point du programme à partir de l'état de sortie de ses prédécesseurs jusqu'à obtention d'un point fixe. Le comportement de chaque instruction vis-à-vis du cache peut ensuite être estimé en fonction du contenu de son état d'entrée.

La *static cache simulation* a été introduite dans un premier temps pour des caches d'instructions à correspondance directe [69] puis étendue pour des caches associatifs utilisant une politique LRU [103, 71]. Dans le cadre des caches de données [103, 104] et unifiés [70], cette famille de méthodes se restreint aux caches à correspondance directe.

L'*interprétation abstraite* [19] fournit le cadre formel de l'autre classe de méthodes prédominant les analyses de cache pour systèmes temps-réel [94]. Trois passes d'analyses, dénommées *Must*, *May* et *Persistence*, sont définies pour calculer respectivement les blocs présents de façon sûre, potentiellement présents, ou non évinçables une fois insérés. De l'analyse d'un cache d'instructions en isolation, ces méthodes ont aussi étendues pour supporter caches de données [25, 87] et unifiés [16].

Dans le cadre de l'analyse des accès aux données se posent des difficultés supplémentaires. En effet, une même instruction dans des contextes différents (pile d'appel, itération de boucle, données d'entrées, etc.) peut accéder à des zones mémoire différentes. La prédiction statique de la cible mémoire d'une telle instruction, lors d'une phase d'*analyse d'adresses,* peut être imprécise. La politique d'écriture est aussi un autre facteur à considérer dans le cadre des analyses. Ces éléments expliquent le besoin d'extensions spécifiques aux travaux existants [103, 104, 25, 87] mais aussi la définition de nouvelles approches [100].

Les *cache miss equations* [100, 98] sont l'une de ces approches spécifiques pour les caches de données. Les nids de boucles de l'application analysée et l'espace d'itérations correspondant sont extraits sous la forme d'un polyèdre dont chaque point correspond à une itération différente [52]. Des vecteurs de réutilisation [106] des blocs mémoire entre les différentes itérations de la boucle peuvent alors être dérivés. La grande précision de cette méthode est toutefois contrebalancée par son domaine d'application restreint. Certains nids de boucles ou schémas d'accès aux données ne rentrent pas dans le cadre du modèle choisi sans transformations nuisant fortement à la précision du modèle dans ce cadre [9].

Ces travaux sont aussi parmi les seuls à considérer les politiques d'écriture en cache. À notre connaissance, seul [26], extension aux méthodes basées sur l'interprétation abstraite [94], offre une estimation des blocs mémoire modifiés et de l'instant de leur éviction du cache. Toutefois, l'imprécision de cette méthode peut conduire à considérer plus d'écritures déportées par *write-back*, lors de l'éviction d'un bloc modifié, que d'instructions d'écritures effectives.

1.2.2.1 Extension du contexte d'applications des analyses de cache

Les méthodes pour l'analyse du contenu de caches d'instructions, de données ou unifiés fournissent des bases pour l'analyse d'une hiérarchie mémoire complète et la prise en compte des différents événements pouvant impacter cette dernière.

Hiérarchies de cache L'analyse d'une hiérarchie de cache pose des problèmes additionnels. En effet, un accès mémoire peut ne pas traverser la totalité de la hiérarchie mais être filtré par les caches des niveaux supérieurs en cas de succès. Le contenu des caches les plus proches de la mémoire dépend donc des succès et des défauts prédits dans les niveaux supérieurs. Les premiers travaux [70] de modélisation de la hiérarchie mémoire se basent sur la classification du comportement succès ou défaut des instructions vis-à-vis de chaque niveau. Durant les analyses ne sont donc modifiés que les niveaux de caches dont l'accès à l'exécution peut être garanti. Pour pallier un problème de sûreté de cette approche, une classification spécifique du comportement des instructions vis-à-vis des niveaux de caches qu'elles accèdent a été proposée [39]. Les différentes politiques de gestion de la hiérarchie, forçant l'inclusion du contenu d'un cache dans ceux de niveaux inférieurs ou l'exclusion des contenus de caches différents, ont ensuite été intégrées dans ce modèle [40]. Les travaux les plus récents [89] préconisent une analyse simultanée de l'intégralité des niveaux de la hiérarchie, en opposition à une analyse niveau par niveau. Cette approche permet de déterminer la position pire cas des blocs à l'échelle de la hiérarchie, c.-à-d. le cache le plus éloigné du processeur où peuvent se trouver les blocs, et non au sein de chaque niveau.

Politiques de remplacement Sur les bases d'une étude théorique [81] visant à évaluer la prédictibilité de différentes politiques de remplacement, les méthodes existantes pour l'analyse de caches ont aussi été étendues afin de supporter des politiques de remplacement autres que le LRU [40]. Une passe d'analyse classique, pour caches LRU, est effectuée en considérant des états de caches abstraits de taille réduite, selon les durées de vie minimum et maximum des blocs dans le cache. Ces durées de vies, ainsi que démontré dans l'étude théorique susmentionnée [81], dépendent de la politique de remplacement du cache. À associativité égale, le LRU est le plus apte à être prédit précisément.

Systèmes préemptifs multi-tâches Les travaux précédents traitent du comportement de la hiérarchie mémoire pour une tâche seule, en isolation ; ces études se concentrent sur les conflits de cache intra-tâche. Dans un système multi-tâche, lors d'une préemption, la tâche qui s'exécutait perd la main sur le processeur et donc sur le contenu du cache. Lorsqu'elle reprend la main, elle peut subir un délai supplémentaire dénommé CRPD (*cache related preemption delay*) lié au rechargement du contenu utile évincé par la ou les tâches préemptantes. Les premières solutions [54] se basent sur l'estimation du volume de l'espace utile, source de dé-

lais pour la tâche préemptée. Un premier raffinement de cette méthode [72] est la prise en compte de l'espace de cache utilisé par la tâche préemptante et son intersection avec cet espace utile. L'approche a par la suite été étendue [90] pour modéliser les préemptions chaînées, liées à la préemption d'une tâche préemptante.

Architectures multi-cœurs Les systèmes multi-cœurs [13] appartiennent à une autre catégorie de systèmes multi-tâches. À la différence des systèmes préemptifs, des tâches peuvent s'exécuter en simultané sur des unités d'exécution séparées. Cette séparation n'est toutefois pas parfaite et certaines ressources sont partagées entre les différents cœurs d'exécution. C'est le cas notamment des derniers niveaux de la hiérarchie mémoire ce qui entraîne cette fois encore l'apparition de conflits inter-tâches. Contrairement aux systèmes préemptifs, ces conflits ne sont pas cantonnés aux seuls points de préemption et se répartissent sur l'intégralité de la durée de vie de chaque tâche.

Un résultat précis peut être obtenu par l'analyse du contenu des caches à partir d'un graphe représentant l'ensemble des entrelacements des accès des tâches concurrentes dans un système. Toutefois, l'obtention et le traitement d'un tel graphe dans le cas général est irréaliste attendu le nombre de possibilités à considérer. En lieu et place, l'estimation, la prise en compte et la réduction du nombre de conflits ont été intégrées pour les caches d'instructions aux méthodes existantes basées sur l'interprétation abstraite [108, 58, 37].

La méthode la plus ancienne [108] repose une première passe d'analyse du comportement de la tâche étudiée vis-à-vis d'un cache à correspondance directe. Les classifications comportementales ainsi obtenues sont raffinées lors d'une étude des accès d'une unique tâche rivale. Par exemple, un accès classifié comme succès dans la tâche analysée est déclassé en défaut s'il existe dans une boucle dans la rivale un accès à un bloc visant le même ensemble. Toutefois, en plus d'un domaine d'application restreint, cette méthode est par trop permissive mettant en jeu la sûreté des estimations obtenues.

Les autres méthodes [58, 37] se basent sur une analyse de la quotité de cache utilisée par chaque tâche concurrente à la tâche analysée. L'espace de cache utilisable de façon sûre par la tâche analysée est réduit d'autant durant la phase d'analyse, jusqu'à devenir nul si les conflits potentiels sont trop nombreux. Les travaux existants s'accordent donc sur le besoin de méthodes de réduction des conflits soit en interdisant le cache à un sous-ensemble des données [37], soit par itérations successives de l'analyse en raffinant l'intersection des temps de vie des tâches pour borner les occurrences de concurrence [58].

1.2.3 Améliorer la prédictibilité des caches

Conflits intra- et inter-tâches peuvent affecter de façon significative et négative la précision des analyses de cache de bas niveau. Partant de ce simple constat, de nombreuses méthodes reposant sur des mécanismes matériels, logiciels, ou hybrides ont été proposées afin de circonscrire ou éliminer ces sources d'indéterminisme. Elles requièrent une intervention logicielle dans la politique de gestion du cache.

Verrouillage

Avec le support matériel du cache, le verrouillage (*locking*) permet de bloquer une partie ou la totalité du contenu du cache ; un sous-ensemble sélectionné des blocs du cache ne peut être choisi pour éviction. Tout accès ultérieur à l'un de ces blocs aboutira forcément à un succès. Toutefois, certaines données utiles peuvent ne pas trouver place dans le cache, si ce dernier est entièrement verrouillé, provoquant ainsi des défauts additionnels.

C'est le cas dans les premières approches utilisant du *locking* pour gagner en prédictibilité dans les systèmes temps-réel [15, 78]. Ces approches figent le contenu du cache d'instructions pour toute la durée de l'exécution d'une tâche. Disposer d'un contenu de cache connu en tout point de programme offre deux avantages pour l'analyse : la localisation précise des succès et des défauts, et un délai de préemption fixe correspondant au coût de chargement du contenu de cache figé.

Des approches plus dynamiques du *locking* [97, 99, 77] proposent la division de chaque tâche en un ensemble de régions, pouvant en couvrir la totalité, pour lesquelles le contenu du cache est figé. Cette composante dynamique permet un regain d'adaptabilité et une circonscription des zones où le cache est figé aux zones de prédictibilité réduite [99].

À l'utilisation, ces méthodes posent toutefois le problème de la sélection du contenu figé dans le cache. Les bénéfices d'un contenu particulier sont relatifs au chemin choisi à l'exécution au sein de la tâche. Une sélection judicieuse conserve des données utiles sous peine de subir d'importantes pénalités. À cette première difficulté s'ajoute celle du découpage de la tâche en régions [77] dans le cadre du *locking* dynamique. Un trop grand nombre de régions implique un coût lié au chargement des données figées en frontière de régions plus grand que les bénéfices du point de vue de la précision.

Une application du *locking* dans le contexte des architectures multi-tâches permet d'éviter les conflits inter-tâches. Plus spécifiquement pour les architectures multi-cœurs, les travaux existants [91] sur de telles applications explorent les dif-

férentes combinaisons de *locking*, statique ou dynamique, en conjonction avec une division du volume de cache par cœur ou par tâche.

Partitionnement

Le partitionnement du cache, sur des bases matérielles ou logicielles, permet la réduction des conflits inter-tâches. Le cache est divisé en partitions, chacune allouée à une tâche du système. Seule la tâche à laquelle appartient une partition peut la modifier en termes de contenu et d'organisation logique du point de vue de la politique de remplacement. L'espace alloué est garanti sans conflit inter-tâches par l'implémentation. De fait, l'analyse de cache peut se focaliser sur le sous-ensemble d'espace alloué à une tâche.

Si les conflits inter-tâches sont réduits par le biais du partitionnement, cette suppression se fait au prix d'une augmentation des conflits intra-tâches. Chaque tâche n'a en effet accès qu'à son sous-ensemble réduit du cache. Qui plus est, de façon similaire au *locking*, des heuristiques sont nécessaires pour décider de l'allocation de l'espace du cache entre les différentes tâches. Pour diriger la recherche d'une solution les méthodes existantes reposent par exemple sur la diminution de la charge globale du système [84] ou la taille de l'empreinte mémoire des tâches [69].

Pour assurer la mise en œuvre du partitionnement ainsi calculé des implémentations logicielles ont été proposées. Elles sont basées sur une modification de la chaîne de compilation [69], compilateur et éditeur de lien, ou du système d'exploitation [60]. L'adresse d'une donnée en mémoire affecte l'ensemble du cache dans lequel cette dernière va être insérée. Chaque tâche est donc placée en mémoire de façon à assurer qu'elle ne pourra accéder qu'à une portion restreinte des ensembles du cache. On parle de partitionnement par ensemble. Ces modifications peuvent toutefois s'avérer coûteuses, notamment pour supporter les caches de données, et difficiles à maintenir. De plus, cette méthode implique que des tronçons de mémoire sont dédiés au bénéfice d'une seule tâche ; une portion de mémoire équivalente à celle qui lui est allouée en cache est réquisitionnée par chaque tâche.

Smart [51] est une implémentation hybride matérielle et logicielle qui permet une division similaire du cache par ensemble. L'attribution des blocs d'une tâche vers les ensembles qui lui sont alloués est assurée par le cache lui-même. L'implémentation matérielle restreint toutefois les tailles valides de partition. Ces dernières doivent être des puissances de deux. L'allocation peut ainsi être implémentée en modifiant simplement la portion d'une adresse mémoire utilisée pour identifier son ensemble de destination. En plus des partitions réservées pour chaque tâche, un

espace commun est alloué dans le cache pour les données partagées et les tâches les moins critiques.

Proposés dans le cadre des systèmes préemptifs multi-tâche, les caches prioritisés [92] (*prioritized-caches*) fonctionnent sur la base d'un pseudo-partitionnement ; la stricte isolation des tâches n'est pas assurée. Une tâche ne peut évincer du cache que les blocs de tâches moins prioritaires ou non critiques. Plus une tâche est prioritaire, moins elle sera sujette aux conflits inter-tâches. Toutefois, cela implique aussi qu'une tâche très prioritaire peut s'approprier la totalité du cache. Dans ce cas, aucune garantie statique en termes d'espace de cache disponible à l'exécution n'existe pour les tâches de faible priorité.

Les travaux présentés dans les derniers paragraphes offrent une vue générale des méthodes ayant trait à l'utilisation ou la mise en œuvre du partitionnement dans le cadre de systèmes temps-réel. De nombreuses études et implémentations différentes ont aussi été proposées pour les systèmes généralistes [50, 107, 80, 73, 48]. Si elles offrent des avantages indéniables dans la réduction des conflits inter-tâches et leur impact sur leurs temps d'exécution moyens, ces méthodes n'offrent pas de garanties statiques suffisantes pour assurer l'isolation des tâches. Les méthodes de pseudo-partitionnements [107, 48] par exemple ont un partitionnement cible mais non garanti. Si d'autres méthodes méthodes [50, 80] permettent de meilleures garanties, elles reposent sur la variation dynamique des tailles de partition. L'avantage de ces méthodes est de reposer sur de simples modifications de la politique de remplacement du cache et en particulier la politique d'éviction [73].

Court-circuitage du cache

Le court-circuitage du cache (*bypass*) [88, 76] est un mécanisme matériel qui permet de restreindre l'accès au cache à certaines données ou instructions. Sur la base par exemple d'indicateurs de localité [45], le bypass permet de garantir que des blocs mémoire ne seront pas insérés en cache et ne perturberont donc pas les données plus utiles. Le mécanisme peut être vu comme le mécanisme inverse du locking.

Pour opérer cette sélection des données à exclure du cache, une première solution [63] consiste à identifier les structures au comportement faiblement prédictible. Cette classification repose sur une analyse qualitative de la nature des accès concernés (scalaires, tableaux, en pile ou sur le tas, etc.) et leur localisation au sein du code de l'application, p. ex. au sein de boucles. Les données sélectionnées sont déplacées dans une zone non-cachable de la mémoire. Cette sélection pure-

ment qualitative peut s'avérer toutefois pessimiste quant à la précision des analyses statiques.

Le bypass a aussi été employé dans le but de réduire l'empreinte de tâches sur le cache [37] avec l'objectif explicite de réduire les conflits inter-tâches en architecture multi-cœur. Cette étude utilise une classification comportementale des instructions obtenue suite à une première phase d'analyse du cache pour restreindre l'accès au cache aux données à l'origine de succès. La précision des analyses de contenu de cache joue ici un rôle décisif dans les résultats de cette sélection. De plus, cette approche cible uniquement les caches d'instructions partagés.

Compilation orientée temps-réel

Des modifications de la chaîne de compilation [24, 62] similaires à celle proposées pour le partitionnement ont aussi été proposées afin d'améliorer la précision des analyses estimant le pire temps d'exécution ou de réduire ce pire temps lui-même. Ces optimisations orientées pour les systèmes temps-réel visent une tâche unique. Par exemple, le placement judicieux des procédures en mémoire peut permettre de réduire les conflits entre fonctions fréquemment voisines dans la pile d'appel [62]. D'autres études ciblent des systèmes entiers afin de réduire les délais de préemption liés au cache en modifiant encore une fois le placement mémoire des tâches [65].

1.3 Discussion

De nombreux travaux ont été effectués à la fin de pouvoir assurer la validation d'un système temps-réel vis-à-vis de ses contraintes temporelles. Ce processus repose sur la précision et la sûreté de l'estimation du comportement temporel des tâches impliquées et principalement l'estimation de leur pire temps d'exécution. Pour une tâche donnée, ce pire cas n'est pas fixe et dépend notamment de l'architecture cible du système, par exemple l'organisation de la hiérarchie mémoire, qui doit donc être prise en compte.

Cette étude porte en priorité sur la hiérarchie mémoire à cause de sa contribution importante au temps d'exécution, *a fortiori* en cas de défauts de cache, même de courtes séquences d'instructions. La majorité des travaux existants [69, 94, 39] porte sur la prise en compte des caches d'instructions dans le cadre de l'analyse de tâches en isolation. Néanmoins, les accès aux données comptent pour une importante part du trafic mémoire des applications. Leur considération est donc

essentielle pour des résultats d'analyse précis.

Les méthodes proposées [103, 25, 104, 87] pour prendre en compte les spécificités des caches de données souffrent d'une précision ou d'une applicabilité limitées. Celles basées sur le modèle polyédrique [100, 98], les plus précises, sont limitées dans leur cadre d'application. De plus, leur combinaison avec des analyses existantes afin de pallier ces limitations n'a pas été étudiée, par exemple en cas de connaissance partielle de la cible en mémoire d'une opération.

De même, les architecture multi-cœurs, répondant à des besoins de performances, ont été peu étudiées dans le cadre de l'estimation temporelle pour systèmes temps-réel. La concurrence entre tâches et le partage de ressources complexifie les analyses ; à tout instant une tâche rivale en agissant sur l'état d'une ressource partagée peut induire des latences additionnelles pour la tâche analysée. De premières études existent pour la prise en compte ou la réduction de ces conflits inter-tâches pour des caches d'instruction partagés [37, 58]. L'applicabilité et l'intérêt de ces méthodes pour les caches de données reste à étudier. Les méthodes développées pour les systèmes multi-tâches et en particulier la réduction du coût de préemption lié aux caches constituent une autre piste d'étude intéressante.

Dans ce même cadre de caches d'instructions partagés, d'autres travaux [91], sur les bases du partitionnement et du *locking*, se concentrent sur la circonscription des conflits. L'avantage indéniable de ces méthodes est de permettre l'utilisation de méthodes d'analyse classiques uni-cœurs sur un espace de cache réduit. De plus, la connaissance de l'intégralité des tâches du système n'est pas requise pour en permettre l'analyse. Toutefois, les mécanismes utilisés peuvent avoir un impact non négligeable sur les performances de tâches non critiques dans le cadre de systèmes de criticité hybride.

Les méthodes présentées et évaluées dans la suite du document visent à pallier ces défauts, notamment la faible représentation des caches de données. Sur la base des analyses bas niveau existantes [40], des méthodes sont proposées pour permettre une prise en compte de hiérarchies mémoire plus riches et la considération de caches de données partagés en contexte multi-cœur (chapitre 2). Enfin, nous présentons une méthode de circonscription des conflits dédiée aux systèmes hybrides, mêlant tâches critiques et non-critiques, et offrant l'avantage de la prédictibilité sans y sacrifier les performances de tâches non critiques concurrentes (chapitre 3).

Chapitre 2

Comportement pire-cas des caches de données partagés en contexte multi-cœur

Introduction

Les caches d'instructions comme de données sont un mécanisme primordial afin d'offrir des performances satisfaisantes pour des systèmes généralistes ou temps-réel. Basés sur les propriétés de localité temporelle et spatiale des tâches, ils permettent de combler le fossé toujours grandissant entre des processeurs de plus en plus rapides et des mémoires principales en comparaison plus lentes. Toutefois, du fait de la nature dynamique de leur comportement, les performances apportées par les caches viennent au prix de leur prédictibilité.

De plus en plus répandues, les architectures multi-cœurs offrent la possibilité d'exécuter concurremment différentes tâches. Le partage de ressources, caches, bus mémoire ou autre, inhérent à ces architectures y est le principal frein à leur prédictibilité ; les occurrences éventuelles de conflits inter-tâches doivent être prises en compte afin d'assurer la sûreté des pires temps d'exécution estimés.

Dans les dernières décennies, de nombreux travaux se sont consacrés à l'estimation de la contribution de hiérarchies mémoire complexes au pire temps d'exécution. Ces études incluent rarement les caches de données dans les hiérarchies considérées et se focalisent sur la contribution des caches d'instructions. Les caches de données posent en effet des problèmes spécifiques tels les écritures mémoire ou l'imprécision de la prédiction statique de l'adresse cible d'une instruction.

Contributions

Les travaux présentés dans cette section permettent l'estimation du comportement temporel au pire cas d'une tâche en présence d'une hiérarchie mémoire comprenant des caches de données. Compatibles avec les extensions aux méthodes existantes [7, 40], p. ex. pour la prise en compte de différentes politiques de remplacement, ils permettent la prise en compte d'un grand nombres de configurations de hiérarchies mémoire [55].

Concernant les caches de données partagés, l'approche proposée [56] est la première pour leur prise en compte dans le cadre d'architectures multi-cœurs. Différentes heuristiques basées sur le mécanisme de court-circuitage de cache (*bypass*), sont aussi introduites. Elles visent à réduire les conflits inter-tâches subis par des tâches concurrentes et ainsi améliorer la précision de l'analyse de niveaux de caches de données partagés.

Organisation

En premier lieu, nous présentons les différentes étapes permettant l'estimation de la contribution temporelle au pire cas d'une hiérarchie de caches de données privés (§ 2.1). Telle qu'esquissée en figure 2.1, cette estimation débute par l'analyse d'adresses, prédiction de la cible des accès aux données en mémoire. L'analyse de caches permet ensuite la classification des accès aux données en tant que succès ou défauts pour chaque niveau de cache. Sur cette base intervient l'estimation de la contribution temporelle de la hiérarchie au pire temps d'exécution. Les méthodes d'analyse d'adresses [6, 86] ne faisant pas partie de notre cadre d'étude, elle ne sont pas détaillées par la suite. L'analyse que nous utilisons est une analyse standard tirée de [38].

Les méthodes d'analyse de cache sont ensuite étendues afin de permettre la prise en compte sûre de niveaux de caches de données partagés tels qu'ils peuvent apparaître sur des architectures multi-cœurs (§ 2.2). Un mécanisme de contrôle des conflits est ensuite étudié afin de réduire l'impact des conflits inter-tâches sur la précision des estimations obtenues (§ 2.3). Les apports et les limites des méthodes proposées sont enfin évalués expérimentalement (§ 2.4).

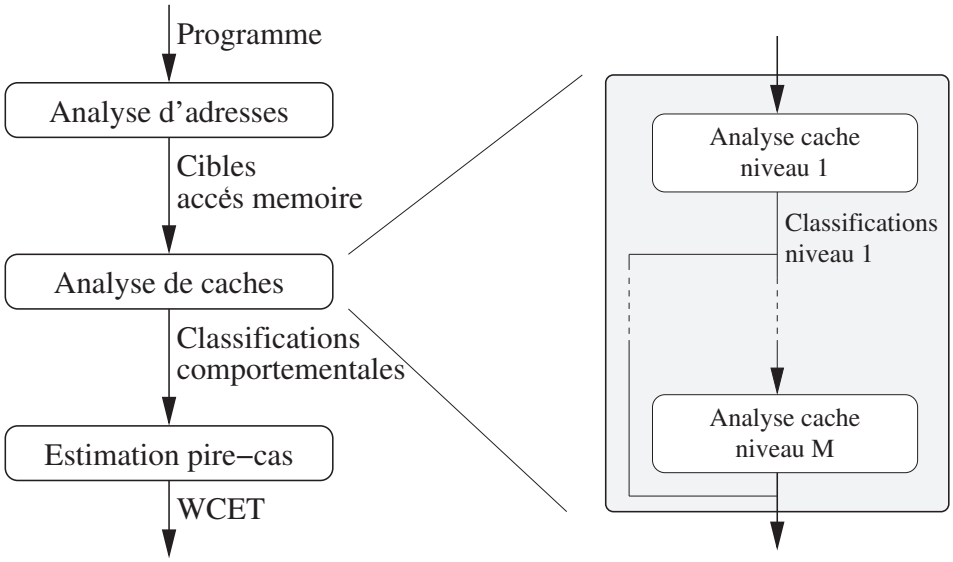


FIGURE 2.1 – Étapes principales requises pour l'estimation de la contribution temporelle au pire cas d'une hiérarchie de caches de données.

2.1 Analyse multi-niveau de caches de données

Le comportement temporel pire cas d'une instruction vis-à-vis de la hiérarchie mémoire dépend des latences qu'elle subit et donc des niveaux de cache auxquels cette dernière accède. L'estimation de ce comportement d'accès requiert la connaissance du comportement succès ou défaut de l'instruction pour chaque niveau. À son tour, cette connaissance repose sur une estimation du contenu à l'exécution des caches de la hiérarchie. Une fois obtenue, la contribution temporelle de l'instruction est intégrée dans une analyse de haut niveau visant à calculer un pire chemin d'exécution.

Les méthodes mentionnées par la suite travaillent à la granularité des références mémoire de l'application analysée. Une référence mémoire est la distinction d'une instruction mémoire dans un contexte d'appel connu. La plage d'adresses ciblée par chaque référence mémoire est estimée statiquement au cours d'une phase d'analyse d'adresses non détaillée dans ce document. Cette plage d'adresses peut, pour des raisons de sûreté, inclure plusieurs blocs mémoire quand la cible exacte d'une référence n'est pas connue statiquement.

De plus, les méthodes présentées sont ici introduites pour des hiérarchies de caches associatifs par ensemble. Celles-ci sont gérées selon une politique de type

mostly-inclusive pour lesquelles ni l'inclusion ni l'exclusion des contenus de caches de niveaux différents ne sont maintenues ; une requête mémoire traverse la hiérarchie jusqu'à trouver la donnée accédée et la charge dans tous les niveaux traversés. Les caches analysés mettent en œuvre une politique de remplacement LRU favorisée pour sa prédictibilité. Les écritures en cache sont supposées propagées à tous les niveaux de la hiérarchie (*write-through*) sans allocation en cache en cas de défaut (*write-no-allocate*).

2.1.1 Analyse d'un niveau de cache

Le comportement temporel pire cas d'une instruction pour un cache donné dépend des défauts et des succès qu'elle y provoque. Un défaut signifie en effet que la donnée doit être servie à une latence plus grande par un niveau de la hiérarchie mémoire plus éloigné du processeur. Pour permettre une telle classification, l'analyse d'un niveau de cache doit estimer les blocs présents dans ce niveau à l'exécution en tout point du programme. L'analyse proposée est une transposition des méthodes proposées dans le cadre des caches d'instruction [94]. Après une présentation de ces méthodes, l'exemple 2.4 en illustre le fonctionnement pour une simple tâche.

Les analyses existantes suivent le schéma classique de méthodes *dataflow* ou basées sur l'interprétation abstraite [19]. De telles analyses reposent sur le calcul en entrée et en sortie de chaque référence mémoire d'une abstraction des états de caches possibles à l'exécution. Cette abstraction est valide quel que soit le chemin emprunté au sein du programme depuis son point d'entrée. L'état de sortie est calculé à partir de l'état d'entrée par une fonction de transfert *(Update)* modélisant l'impact de la référence mémoire sur le contenu de cache. L'état d'entrée est la conjonction des états de sortie des prédécesseurs de la référence dans le programme. Cette fonction de conjonction *(Join)* intervient en cas de convergence de flot, p. ex. une instruction en sortie de conditionnelle peut avoir plusieurs prédécesseurs.

Les accès non déterministes, dont la cible mémoire n'est pas déterminée de façon précise statiquement, sont une spécificité des caches de données par rapport aux caches d'instructions. Ces accès ne sont donc pas modélisés par les méthodes classiques [94]. La plage d'adresses attachée à une référence mémoire par l'analyse d'adresses représente autant d'alternatives, *de chemins possibles,* pour le choix du bloc accédé par la référence à l'exécution. Nous divisons le calcul de l'état de cache en sortie d'un accès non déterministe en deux étapes. Pour chacun des blocs de cache auquel la référence peut accéder, nous calculons l'état de cache résultant de cet accès grâce à la fonction de transfert *Update*. La conjonction de ces différents

états possibles par le biais de la fonction *Join* permet ensuite de dériver l'état de cache en sortie de la référence mémoire non déterministe.

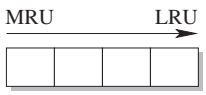
Trois passes d'analyses distinctes ont été définies dans [94]. Chacune repose sur une fonction de transfert *(Update)* et une fonction de jointure *(Join)* qui lui est propre. Les blocs présents dans les états abstraits calculés par chaque analyse répondent à une propriété particulière. L'analyse *Must*, base de l'exemple 2.4, détermine les blocs présents de façon sûre à l'exécution. L'analyse *May* capture ceux qui peuvent être présents. L'analyse de *Persistence* capture les blocs persistants en cache au sein des boucles, ceux qui une fois insérés ne peuvent être évincés par les itérations successives d'une boucle. Notons que notre modélisation des accès non déterministes est indépendante des fonctions *Join* et *Update* sous-jacentes. Elle s'applique donc de façon identique aux trois phases d'analyse.

Du contenu du cache calculé par chaque passe, une classification de succès ou défaut (CHMC, *cache hit/miss classification*) est dérivée pour chaque référence mémoire selon qu'elle provoque toujours un succès en cache (AH, *always-hit*, capturé par l'analyse *Must*), un défaut (AM, *always-miss*, analyse *May*), que son comportement soit inconnu (NC, *not-classified*), ou que le premier accès à chacun de ses blocs cibles ait un comportement inconnu mais que les suivants soient des succès (FM, *first-miss*, analyse *Persistence*). La classification FM implique que le nombre de défauts subis par une référence mémoire est borné par le nombre de blocs différents auxquels elle peut accéder.

EXEMPLE 2.4 – Analyse de contenu de cache

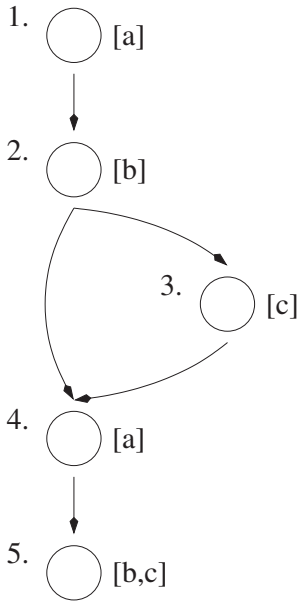
Nous présentons ici le déroulement d'une analyse de contenu *Must* dont l'objectif est de déterminer les blocs présents de façon sûre en cache à l'exécution. Cette analyse permet la classification d'une référence en tant que succès (AH) si les blocs qu'elle cible sont présents dans son état de cache en entrée. Par défaut, une classification NC est supposée si le succès en cache ne peut être garanti.

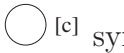
L'analyse est illustrée pour un simple cache entièrement associatif de 4 voies. Les blocs de ce dernier sont présentés selon leur âge logique du plus récemment utilisé (MRU) au moins récemment utilisé (LRU) :



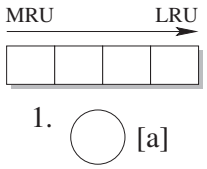
La tâche considérée durant l'analyse est composée d'une simple conditionnelle. Elle comprend 5 références mémoire seules représentées ici. Les arêtes symbolisent

les flots possibles dans le programme :

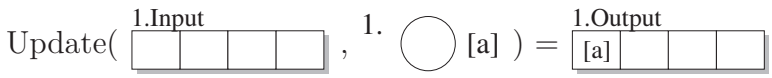


Où  symbolise la référence mémoire 3 qui accède au bloc $[c]$. La référence mémoire 5 est non déterministe et peut accéder au bloc $[b]$ ou au bloc $[c]$ à l'exécution.

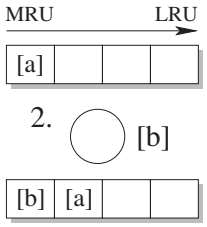
Référence mémoire 1 Nul bloc ne peut être garanti comme présent en cache au démarrage ; l'état de cache en entrée de la référence mémoire 1 est vide :



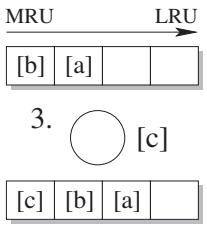
Suite à l'exécution de la référence 1, le bloc $[a]$ est inséré en tête du cache en position MRU. Afin de modéliser cet impact sur le cache et donc calculer l'état du cache en sortie de la référence mémoire, la fonction de transfert *Update* est utilisée :



Référence mémoire 2 La référence mémoire 2 a pour seul prédécesseur la référence 1. L'état du cache en entrée de 2 est donc l'état du cache en sortie de 1. Similairement à la référence 1, 2 insère le bloc auquel elle accède, $[b]$, en position MRU et repousse les blocs présents vers la position LRU :

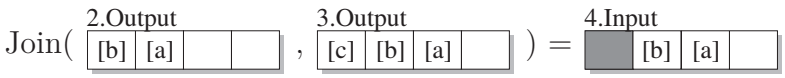


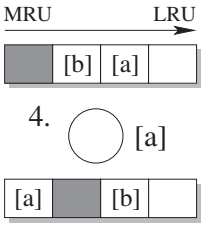
Référence mémoire 3 La référence 3 n'a qu'un unique prédécesseur et accède à un bloc identifié précisément, $[c]$. Le calcul de ses états de cache en entrée et en sortie suit le principe évoqué précédemment :



Référence mémoire 4 L'état de cache en entrée du point 4 doit regrouper l'information disponible en sortie de ses prédécesseurs 2 et 3. Dans le cadre de l'analyse *Must* qui vise à déterminer les blocs présents de façon sûre en cache, le *Join* de deux états de cache calcule l'intersection des blocs présents et les positionne en cache à leur âge maximum. À titre de comparaison, l'analyse *May* calcule l'union des blocs présents dans les deux caches et leur âge minimum. L'analyse *Persistence* calcule l'union des blocs présents et évincés, et leur âge maximum.

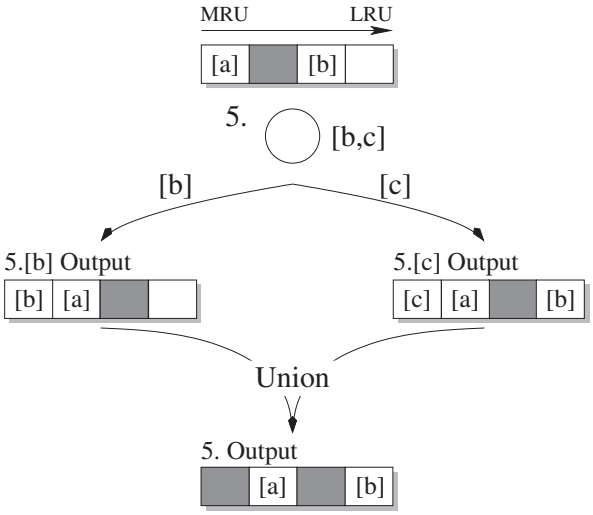
Par exemple, $[c]$ n'est présent en cache que si le chemin passant par la référence 3 est emprunté à l'exécution ; $[c]$ peut être absent du cache avant l'exécution de 4. $[a]$ et $[b]$ sont forcément présents en cache à ce point de programme, mais ils peuvent avoir été poussés vers la position LRU lors de l'insertion de $[c]$ en cache. L'état d'entrée en 4 reflète ces comportements :





Le bloc $[a]$ auquel la référence 4 accède est garanti être présent en cache avant son exécution. Elle peut être classifiée comme étant un succès par l'analyse *Must*.

Référence mémoire 5 La cible exacte de la référence mémoire 5 n'a pu être déterminée par l'analyse d'adresses. Du point de vue statique, l'instruction peut accéder au bloc $[b]$ ou $[c]$ à l'exécution. Nous considérons la conjonction des états résultant des différentes alternatives en termes de bloc mémoire accédé :



La référence 5 ne peut être classifiée en tant que succès. Si le bloc $[b]$ est garanti présent en cache avant exécution, une telle garantie n'existe pas pour le bloc $[c]$. Du point de vue de l'analyse, il existe donc une possibilité que la référence soit un défaut de cache.

2.1.2 Classification d'accès au cache (CAC)

Lors d'un accès mémoire, une instruction n'accède pas à tous les niveaux de la hiérarchie mémoire. Les accès sont filtrés par le premier niveau où est trouvée

la donnée recherchée. Ce filtrage doit être pris en compte lors des analyses de contenu de cache afin d'estimer les niveaux modifiés par chaque référence mémoire. Statiquement, pour savoir si un niveau donné est accédé par une référence mémoire, il faut estimer la classification succès ou défaut (CHMC) mais aussi le comportement d'accès de cette référence au niveau précédent de la hiérarchie mémoire. Puisque nous reposons sur une extension aux caches de données des méthodes d'analyse pour caches d'instructions, les concepts sont identiques à ceux définis par [39] pour des hiérarchies de caches d'instructions.

Une classification supplémentaire (CAC, *cache access classification*) de chaque instruction est requise pour identifier les niveaux qu'elle accède à chaque fois (A, *always*), n'accède jamais (N, *never*), peut accéder ou non (U, *uncertain*), peut accéder ou non lors de ses premières occurrences au sein d'une boucle mais n'accède pas par la suite (UN, *uncertain-never*). Cette classification est obtenue à partir de la CHMC et la CAC du niveau précédent ainsi qu'illustré dans le tableau 2.1. Par exemple, si l'on peut garantir un succès au niveau $L-1$ avec la classification AH, alors le niveau L ne sera jamais accédé (classification N) par la référence mémoire. Cela impose l'analyse des niveaux de la hiérarchie dans l'ordre depuis le processeur jusqu'à la mémoire (illustré en figure 2.1 p. 27).

		$\text{CHMC}_{r,L-1}$			
		AH	FM	AM	NC
$\text{CAC}_{r,L-1}$	A	N	UN	A	U
	N	N	N	N	N
	UN	N	UN	UN	UN
	U	N	UN	U	U

TABLE 2.1 – Calcul des classifications d'accès au cache de niveau L pour la référence mémoire r ($\text{CAC}_{r,L}$).

La CAC d'une instruction, une fois calculée à partir des informations du niveau précédent, est prise en compte dans l'analyse de cache d'un niveau par la fonction de mise à jour de contenu de cache *(Update)*. La CAC de chaque instruction est initialisée à A pour le premier niveau de la hiérarchie celui-ci étant toujours interrogé en premier lors d'un accès mémoire.

2.1.3 Estimation de la contribution temporelle au pire cas des caches

La contribution temporelle de la hiérarchie mémoire au pire temps d'exécution dépend du comportement de chaque instruction vis-à-vis d'icelle. Chacun des niveaux de cache accédés par une instruction va induire des latences supplémentaires. Ces latences et donc la contribution d'une instruction au pire temps d'exécution sont estimées à partir des classifications d'accès (CAC) et comportementales (CHMC) dérivées durant l'analyse de cache. On se place dans le cadre d'une architecture sans anomalies temporelles [64] (*timing anomalies*) où un défaut constitue le pire cas du point de vue temporel.

Par la suite, nous nous focalisons sur l'estimation de la contribution temporelle pire cas des références mémoire. Cette contribution est intégrée dans les méthodes existantes basées sur l'énumération implicite de chemin [59] (IPET, *implicit path enumeration technique*). Ces méthodes reposent sur la résolution d'un système de contraintes ILP, *integer linear programming* [85] représentant le flot de contrôle dans le programme. Le temps d'exécution du programme est ainsi modélisé et maximisé par une fonction objectif. Sauf mention contraire, quand le coût d'une instruction ou référence mémoire est mentionné nous signifions par là sa contribution temporelle vis-à-vis de la hiérarchie mémoire et en particulier des caches de données.

Au sein d'une boucle, lors d'itérations successives, le coût d'une référence mémoire peut diminuer à mesure qu'elle charge en cache les blocs auxquels elle accède. De fait, pour chaque référence mémoire r, on distingue $COST\_first(r)$ et $COST\_next(r)$ représentant respectivement son coût lors de la première itération de la boucle la plus externe incluant r, et son coût lors des itérations suivantes. Le nombre d'occurrences de chacun de ces scénarios sur le pire chemin d'exécution, dénommés respectivement $freq_{first,r}$ et $freq_{next,r}$, est calculé par l'IPET. La contribution totale de la référence r au pire temps d'exécution s'exprime alors ainsi :

$$\text{WCET}_{data}(r) = COST\_first(r) \times freq_{first,r} + COST\_next(r) \times freq_{next,r}$$

$COST\_first(r)$ et $COST\_next(r)$ sont des constantes du problème IPET. Elles sont calculées à partir des résultats de l'analyse de la hiérarchie mémoire, c'est à dire les classifications CAC et CHMC de la référence mémoire r vis-à-vis de

chacun des niveaux de la hiérarchie. $freq_{first,r}$ et $freq_{next,r}$ à l'inverse sont des variables dont la valeur est dérivée lors de la résolution du problème IPET.

Pour déterminer $COST\_first(r)$ et $COST\_next(r)$, les niveaux de caches sont répartis en trois catégories distinctes : ceux qui ne contribuent jamais, identifiés par une classification d'accès N ($\text{CAC}_{r,L} = \text{N}$) ; ceux qui contribuent dans le pire cas à la latence de chaque accès de la référence r, $always\_contribute(r)$; et ceux qui ne contribuent que lors des premiers accès de r à chacun de ses blocs, $first\_contribute(r)$.

$always\_contribute(r)$ comprend les niveaux de cache dont la classification d'accès est certaine (A) ou possible (U). De plus, une classification FM au niveau $L-1$ implique un filtrage des accès au niveau L après les premières occurrences de r. Toutefois, si r cible un nombre de blocs différents plus grand que son nombre maximum d'occurrences, il est conservatif de supposer que chaque accès de r au niveau $L-1$ est un défaut et le niveau L toujours accédé. En posant $|memory\_blocks_{r,L-1}|$, ce nombre de blocs cible de r au niveau $L-1$ et max_freq_r, borne sur le nombre d'occurrences de r, on peut définir $always\_contribute(r)$ ainsi :

$$
\begin{aligned}
&always\_contribute(r) = \\
&\{L \mid 1 \leq L \leq M \wedge \text{CAC}_{r,L} = \text{A}\} \bigcup \{L \mid 1 \leq L \leq M \wedge \text{CAC}_{r,L} = \text{U}\} \bigcup \\
&\{L \mid 1 \leq L \leq M \wedge \text{CAC}_{r,L} = \text{UN} \wedge (L-1) \in always\_contribute(r) \wedge \text{CHMC}_{r,L-1} \neq \text{FM}\} \bigcup \\
&\{L \mid 1 \leq L \leq M \wedge \text{CAC}_{r,L} = \text{UN} \wedge (L-1) \in always\_contribute(r) \wedge \text{CHMC}_{r,L-1} = \text{FM} \\
&\wedge |memory\_blocks_{r,L-1}| > max\_freq_r\}
\end{aligned}
$$

$|memory\_blocks_{r,L-1}|$ est obtenu à partir des résultats de l'analyse d'adresses qui associe à chaque référence une estimation de sa cible en mémoire. max_freq_r peut être estimé simplement comme le produit des bornes d'itération des boucles englobant r.

Par opposition, $first\_contribute(r)$, qui identifie l'ensemble des niveaux de caches accédés uniquement lors du premier accès de r à chacun de ses blocs, s'exprime ainsi :

$$
\begin{aligned}
&first\_contribute(r) = \\
&\{L \mid 1 \leq L \leq M \wedge L \notin always\_contribute(r) \wedge \text{CAC}_{r,L} \neq \text{N}\}
\end{aligned}
$$

$COST\_next(r)$ est défini comme la somme des latences d'accès des niveaux de cache dont la contribution au coût de r est certaine, c'est à dire les niveaux de cache appartenant à $always\_contribute(r)$:

$$COST\_next(r) = \begin{cases} \sum_{L \in\ always\_contribute(r)} ACCESS\_latency_L & \text{si r est une lecture} \\ STORE\_latency & \text{si r est une écriture} \end{cases}$$

La définition de $COST\_first(r)$ doit prendre en compte ces mêmes niveaux dont la contribution est certaine, mais aussi les niveaux accédés lors des premiers accès de r. Il est considéré que r charge l'intégralité de ses blocs persistants d'un seul tenant. Les blocs persistants sont ceux qui une fois insérés au sein d'une boucle ne sont pas évincés lors d'itérations successives. Les latences d'accès aux niveaux de $first\_contribute(r)$ ne sont considérées que lors de la première occurrence de r :

$$COST\_first(r) = \begin{cases} \sum_{L \in\ always\_contribute(r)} ACCESS\_latency_L + & \text{si r est une lecture} \\ \sum_{L \in first\_contribute(r)} ACCESS\_latency_L \times max\_occurrence(r, L) & \\ STORE\_latency & \text{si r est une écriture} \end{cases}$$

$max\_occurrence(r, L)$ est une borne statique sur le nombre d'occurrences de la référence mémoire r au niveau de cache L. Dans le cadre des niveaux de cache appartenant à l'ensemble $first\_contribute(r)$, elle borne le nombre de premiers accès vers ces niveaux :

$$max\_occurrence(r, L) = \begin{cases} 0 & if\ \text{CAC}_{r,L} \neq \text{N} \\ max\_freq_r & if\ L \in always\_contribute(r) \\ |memory\_blocks_{r,L-1}| & if\ \text{CHMC}_{r,L-1} = \text{FM} \\ max\_occurrence(r, L-1) & otherwise \end{cases}$$

Une borne plus précise que max_freq_r du nombre d'occurrences de la référence r est $freq_r$, la fréquence d'exécution de r sur le pire chemin d'exécution de la tâche. Toutefois, le calcul de cette borne $freq_r$ est requis pour obtenir la contribution de r au pire temps d'exécution et inversement, menant ainsi au paradoxe de l'œuf et de la poule [27].

2.2 Impact des interférences sur les analyses de caches de données

Les méthodes présentées précédemment visent à estimer la contribution temporelle au pire cas de hiérarchies de caches de données. Ces méthodes reposent sur l'estimation du contenu des caches en chaque point du programme puis la classification du comportement des références mémoire d'une application vis-à-vis de ces caches. Les analyses de caches de données supposent toutefois l'isolation de la tâche analysée en termes de contenu de cache.

Dans le cadre d'architectures multi-cœurs, certains niveaux de cache peuvent être partagés entre différents cœurs et ainsi se retrouver modifiés par des tâches concurrentes. Ces modifications du cache sont indépendantes de la tâche analysée. Un contenu de cache estimé sans considérer les conflits inter-tâches met donc en jeu la sûreté des estimations temporelles qui en découlent.

Nous présentons par la suite des méthodes permettant l'estimation et la prise en compte des conflits inter-tâches subis par une tâche pour un niveau de chaque cache partagé L quelconque. Dans les faits, ces méthodes doivent être appliquées pour chaque niveau de cache partagé.

2.2.1 Estimation des conflits de cache inter-tâches

Le calcul des entrelacements entre accès concurrents permettrait d'estimer l'impact des interférences sur le contenu de la hiérarchie mémoire, mais s'avère trop coûteux dans le cas général. Pour réduire ce coût, abstraction est faite du nombre d'occurrences et de l'ordre des accès des tâches rivales ; une tâche concurrente est supposée accéder à tout instant à n'importe lequel de ses blocs un nombre indéfini de fois. Pour chaque ensemble du cache partagé considéré, le calcul des interférences générées par les tâches rivales se résume donc à compter le nombre de blocs en conflit avec ceux de la tâche analysée. Il s'agit d'une estimation sûre bien que pessimiste du nombre de lignes du cache occupées par les blocs de tâches rivales.

Le nombre de conflits que la tâche i subit dans l'ensemble s du cache partagé L est dénoté $\text{CCN}_{i,L}(s)$ (*cache conflicts number*). Il correspond au nombre de blocs différents auxquels ses rivales peuvent potentiellement accéder au niveau de cache L. Ces blocs peuvent être identifiés en utilisant la classification CAC :

$$\text{CCN}_{i,L}(s) = |\bigcup_{u \in T} task\_blocks_{u,L}(s)|$$

$$\begin{aligned} task\_blocks_{u,L}(s) = \{b \mid \exists r \in memory\_references_u \wedge \\ b \in memory\_blocks_{r,L} \wedge \text{CAC}_{r,L} \neq \text{N} \wedge \\ CacheSet_L(b) = s\} \end{aligned}$$

Où T représente l'ensemble des tâches du système étudié potentiellement concurrentes à la tâche i ; $memory\_references_u$ l'ensemble des références mémoire de la tâche u rivale de i ; et $memory\_blocks_{r,L}$ l'ensemble des blocs mémoire accédés par la référence r au niveau de cache L tel qu'estimé par l'analyse d'adresses. Un $\text{CAC}_{r,L} \neq N$ implique que la référence r de la tâche rivale peut être cause du chargement de blocs mémoire dans le cache L ; $CacheSet_L(b) = s$ que s est l'ensemble de destination du bloc b en cache.

2.2.2 Prise en compte des conflits durant l'analyse des caches

Le CCN d'une tâche est une représentation des conflits générés par ses rivales sans information d'ordre ou d'occurrences des accès. Si tous les accès concurrents possibles sur le cache partagé ont lieu, ils ne peuvent insérer ou promouvoir plus de CCN blocs différents en cache. Dans ce scénario pire cas, sous la politique de remplacement LRU supposée (p. 27), les blocs de la tâche analysée sont repoussés d'autant vers l'éviction. L'estimation des conflits donnée par le CCN permet donc de borner le nombre de lignes de cache occupées par des tâches rivales.

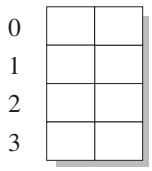
L'occupation du cache par les blocs de tâches rivales doit être prise en compte par l'analyse de cache partagé. Pour ce faire, durant cette phase d'analyse, la carte des conflits est appliquée aux états de cache manipulés ; seules les lignes garanties comme utilisables par la tâche analysée sont examinées. Étant donnée $Associativity_L$, l'associativité du cache partagé analysé, pour chaque ensemble s du cache, $max(Associativity_L - \text{CCN}_{i,L}(s), 0)$ lignes peuvent être utilisées de façon sûre par la tâche i. En dehors de l'associativité des états de caches manipulés, la phase

d'estimation de contenu de cache et la phase de classification comportementale restent inchangées.

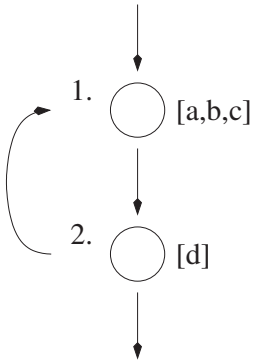
L'analyse d'une hiérarchie de caches est toujours effectuée niveau par niveau depuis le plus proche du processeur au plus éloigné. L'analyse d'un niveau privé est inchangée. Elle requiert l'obtention des classifications CAC et CHMC de la tâche considérée au niveau précédent. L'analyse d'un niveau partagé requiert en plus ces classifications pour les tâches rivales à celle analysée. Pour chaque niveau de cache partagé, l'analyse doit donc être effectuée pour toutes les tâches avant de progresser au niveau suivant. Cet ordonnancement garantit que l'on dispose des classifications comportementales requises pour l'analyse du niveau suivant du point de vue de chaque tâche elle-même mais aussi de ses rivales.

EXEMPLE 2.5 – Calcul et prise en compte des interférences

Cet exemple a pour but d'illustrer l'estimation des conflits subis par une tâche *analysée* et leur prise en compte. À cette fin, supposons un unique cache L1 partagé composé de 4 ensembles de 2 voies chacun :



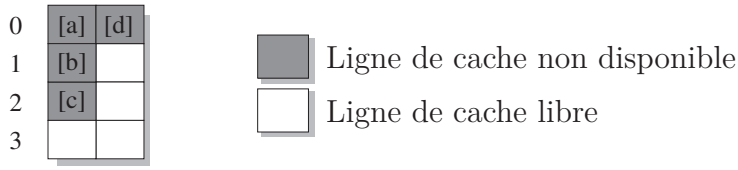
La tâche *analysée* est opposée à une unique tâche *concurrente*, seule à contribuer aux conflits inter-tâches sur le cache partagé. Cette *concurrente* comprend deux références mémoire réparties au sein d'une boucle et accédant respectivement aux blocs $[a]$, $[b]$ ou $[c]$, et au bloc $[d]$:



On suppose que les blocs $[a]$, $[b]$, $[c]$ et $[d]$ se retrouvent respectivement dans les ensembles 0, 1, 2 et 0 du cache. L'estimation des conflits générés par la tâche *concurrente* est la suivante :

$$task\_blocks_{concurrente,1} = \begin{cases} 0 \rightarrow \{a, d\} \\ 1 \rightarrow \{b\} \\ 2 \rightarrow \{c\} \\ 3 \rightarrow \emptyset \end{cases}$$

L'espace de cache considéré durant l'analyse de la tâche *analysée* est réduit en conséquence ; chaque bloc chargé par la *concurrente* peut occuper une ligne du cache :



Une approche sûre est de considérer que la tâche *analysée* ne peut accéder à l'ensemble 0 du cache partagé et à une seule ligne dans les ensembles 1 et 2. Les deux lignes de l'ensemble 3, garanties libres de conflits, lui sont accessibles.

2.2.3 Classification des accès aux données partagées

Les données partagées entre différentes tâches requièrent des mesures particulières. Durant l'exécution de la tâche analysée, un bloc de données partagées peut en effet être inséré dans le cache par une tâche rivale. Afin de modéliser cet effet constructif, les références mémoire vers de tels blocs ne peuvent être classifiées comme étant toujours en défaut (AM) pour les caches partagés. La classification NC est utilisée en lieu et place.

Du fait des protocoles de cohérence [4], une donnée partagée peut se voir invalidée dans un cache, privé ou partagé, sur requête d'une tâche concurrente. Dans le cas général, la classification d'accès aux données partagées en tant que succès, AH ou FM, dans les niveaux de cache privés et partagés ne peut être garantie. Néanmoins, l'insertion et la promotion de ces blocs doivent être modélisées par les fonctions de transfert durant la phase d'analyse des caches.

La classification en succès, AH ou FM, d'une référence mémoire à des données partagées peut être garantie dans l'une ou l'autre de ces configurations particulières :

- Le protocole de cohérence repose sur la mise à jour des caches voisins en cas d'écriture *(write-update)*, par opposition à l'invalidation *(write-invalidate)*. En contrepartie, cette famille de méthodes [93, 5] s'accompagne d'un trafic inter-cœurs plus important.
- Les caches des derniers niveaux, les plus proches de la mémoire, sont accessibles par tous les coeurs où peuvent s'exécuter les tâches partageant des données. Les blocs stockés dans ces niveaux ne peuvent être la cible d'invalidations.
- La tâche analysée ne peut être exécutée concurremment à celles modifiant les données ciblées. Si invalidation il y a, elle intervient en dehors de la durée de vie de la dite tâche. Une telle assurance requiert la considération de l'ordonnancement du système ou la synchronisation des lecteurs et des rédacteurs de la donnée partagée.

2.3 Réduction des conflits pour les caches partagés, utilisation du bypass

L'estimation des interférences subies par une tâche repose sur l'abstraction des informations disponibles concernant les tâches rivales, en particulier l'ordre et les occurrences de leurs accès au cache partagé. Ces abstractions assurent l'efficacité de l'analyse de conflits dans le cas général au prix d'un certain pessimisme. Pour des raisons de sûreté, l'ensemble des lignes du cache peut devoir être considéré comme occupé par des tâches rivales.

Le court-circuitage du cache (bypass) est un mécanisme permettant d'empêcher la modification du contenu d'un cache par un accès mémoire. Lors de cet accès, le niveau de cache sous bypass est traversé pour y chercher la donnée, mais nulle modification de l'état logique du cache, promotion ou insertion, n'a lieu. Le bypass est par exemple présent sous la forme d'une décision par instruction dans les architectures Intel IA-64 [45]. À l'inverse dans le PowerPC 403GA [44], décision peut être prise par donnée en interdisant l'entrée en cache de certains segments mémoire.

Si un accès mémoire court-circuite un niveau de cache, il ne contribue pas aux conflits intra- ou inter-tâches à ce niveau de la hiérarchie. Nous utilisons cette propriété pour réduire les conflits sur les caches partagés dans le cadre de systèmes multi-cœurs. À cette fin, nous présentons des méthodes permettant pour chaque instruction de définir si elle doit ou non court-circuiter un cache partagé donné.

L'obtention d'une solution optimale, au regard par exemple de l'utilisation du système, s'avère trop complexe dans le cas général. Les approches proposées sont donc heuristiques. Pour chaque instruction, elles reposent sur l'évaluation de la réutilisation faite des blocs qu'elle charge en cache, ou d'aspects qualitatifs sur la nature des données qu'elle cible.

2.3.1 Calcul des réutilisations entre accès mémoire

Les références mémoire accédant à un même bloc mémoire sont réparties en différents points de la tâche analysée. Évaluer l'intérêt de la contribution au cache d'une référence mémoire requiert d'identifier les références ultérieures qui peuvent utiliser les blocs mémoire qu'elle y charge ou promeut. En l'absence de réutilisation, la contribution de la référence mémoire considérée est dispensable et elle peut court-circuiter le cache sans impact.

Pour une référence mémoire, le processus repose sur le calcul de l'ensemble des références mémoire qu'elle peut impacter par sa contribution au cache, c'est à dire celles : qui accèdent au même bloc ; qui sont accessibles par un chemin dans l'application ; et telles qu'il n'existe pas de référence à ce même bloc sur le chemin de l'une à l'autre. On identifie ainsi l'ensemble $next\_load_L(inst)$ des instructions qui utilisent un bloc inséré ou rafraîchi dans le cache L par l'instruction $inst$:

$$next\_loads_L(inst) = \{r' \mid \exists r, b, r \in memory\_references(inst) \wedge b \in memory\_blocks_{r,L} \wedge r' \in next\_reference_L(r, b)\}$$

Où $memory\_references(inst)$ désigne l'ensemble des références mémoire liées à $inst$, une pour chacun de ses contextes d'exécution ; $memory\_blocks_{r,L}$ la plage de blocs mémoire accédés par la référence r sur le cache de niveau L. Les références mémoire dépendantes sont liées entre elles par $next\_reference_L(r, b)$.

$next\_reference_L(r, b)$ est défini comme l'ensemble des références liées à la référence r par un bloc b commun. Pour rappel, dans les hiérarchies considérées (p. 27), les écritures sont propagées à tous les niveaux de la hiérarchie sans allocation en cache en cas de défaut. Les écritures mémoire n'interviennent donc pas dans le calcul des dépendances. Une référence mémoire r' appartient à $next\_reference_L(r, b)$ si elle peut accéder au bloc b et si il existe un chemin de r à r' sans autre référence à b. Plus formellement :

$$next\_reference_L(r, b) = \{r' \mid b \in memory\_blocks_{r',L} \wedge r' \text{ is a load} \wedge r' \in successors(r, \text{CFG}_t \triangleright b)\}$$

Où $successors(n, G)$ est l'ensemble des successeurs du nœud n dans le graphe G. $\text{CFG}_t \triangleright b$ est le CFG, *control flow graph* de la tâche t réduit aux références au bloc mémoire b. $\text{CFG}_t \triangleright b$ peut être construit en retirant du CFG de la tâche t les blocs de base puis les instructions qui ne sont pas des références mémoire au bloc b. Les arêtes du graphe sont conservées par transitivité. Dans une telle construction, les accès non déterministes constituent autant de chemins possibles que de blocs différents auxquels ils peuvent accéder.

2.3.2 Heuristiques pour le bypass

Pour une instruction, le choix de court-circuiter ou non un niveau de cache n'est pas anodin. Cette décision entraîne des répercussions sur le comportement des références mémoire suivantes. Certaines peuvent devenir des défauts si elles utilisaient le bloc inséré par l'instruction. Mais à l'inverse, si l'instruction en court-circuit participait à l'éviction d'un bloc utile, certaines références mémoire peuvent se résoudre en succès suite au bypass. Explorer l'ensemble des configuration de bypass de chacune des instructions du programme, pour sélectionner la meilleure, est toutefois trop complexe.

En lieu et place, nous définissons des heuristiques permettant d'atteindre pour un niveau de cache donné une décision locale à chaque instruction de chargement mémoire. Les deux premières heuristiques reposent sur l'estimation de la réutilisation des blocs chargés par l'instruction. La troisième heuristique repose sur les résultats de l'analyse d'adresses et la nature des structures accédées.

Pour une heuristique x, la décision de bypass pour l'instruction *inst* et le niveau de cache L est formalisée par $Bypass_x(inst, L)$. $Bypass_x(inst, L)$ est $true$ (vrai) si l'instruction *inst* court-circuite le niveau de cache L selon l'heuristique x. À l'inverse, $Bypass_x(inst, L) = false$ si l'instruction *inst* ne court-circuite pas le niveau de cache L.

Conservative bypass. En utilisant l'heuristique de bypass conservative (CB, *conservative bypass*), si les blocs chargés par l'instruction *inst* au niveau de cache L *peuvent* être réutilisés avant éviction, alors l'instruction ne court-circuite pas ce niveau de cache :

$$Bypass_{CB}(inst, L) = (\forall r \in next\_loads_L(inst), \text{CHMC}_{r,L} \neq \text{AH} \wedge \text{CHMC}_{r,L} \neq \text{FM})$$

Cette heuristique est conservative en ce qu'elle choisit de conserver la contribution d'une instruction si elle est à l'origine d'au moins un succès en cache capturé

par les analyses statiques.

Aggressive bypass. L'heuristique agressive (AB, *aggressive bypass*) se place dans une optique opposée à celle de CB. Une instruction *inst* court-circuite le niveau de cache L si les blocs qu'elle charge peuvent être évincés avant réutilisation :

$$Bypass_{AB}(inst, L) = (\exists r \in next\_loads_L(inst), \text{CHMC}_{r,L} = \text{AM} \vee \text{CHMC}_{r,L} = \text{NC})$$

Indeterministic bypass. Le bypass *non déterministe* (IB, *indeterministic bypass*) ne repose pas sur le calcul des relations *next_loads* entre instructions. Une instruction *inst* court-circuite le niveau de cache L si l'analyse d'adresses n'a pu déterminer précisément le bloc accédé par cette dernière ; IB cible toutes les instructions provoquant des accès non déterministes dans le cadre des analyses :

$$Bypass_{IB}(inst, L) = (\exists r \in memory\_references(inst), |memory\_blocks_{r,L}| > 1)$$

2.3.3 Analyse de statique caches de données avec du bypass

L'utilisation du court-circtuitage au sein d'une application modifie son comportement vis-à-vis du cache ciblé. Les changements induits doivent être pris en compte lors d'une passe d'analyse du cache. Cette passe d'analyse doit modéliser le mécanisme de bypass et en particulier l'impact d'une instruction court-circuitant le cache sur son contenu.

Une instruction qui court-circuite un cache n'a d'impact ni sur les blocs contenus dans ce dernier ni sur leur organisation logique ; l'état du cache en sortie d'une référence mémoire qui le court-circuite est identique à son état d'entrée. La fonction de transfert *Update* utilisée dans le cadre des analyses de contenu de cache est modifiée afin de modéliser correctement ce comportement. Une référence même si elle court-circuite un niveau de cache doit subir le processus de classification CHMC et CAC vis-à-vis de ce dernier. La donnée accédée peut avoir été chargée par un accès antérieur et filtrer les accès vers le niveau suivant.

2.4 Expérimentations

Après une présentation des conditions d'expérimentations (§ 2.4.1), nous évaluons la précision des méthodes proposées pour l'estimation de la contribution au pire temps de caches de données. Nous nous plaçons tout d'abord dans un contexte uni-cœur (§ 2.4.2) avec un seul niveau de cache puis une hiérarchie à deux niveaux. Un cache partagé est ensuite inclus dans la hiérarchie analysée afin d'estimer l'impact des conflits inter-tâches et du bypass dans le cadre d'architectures multi-cœurs (§ 2.4.3).

2.4.1 Conditions expérimentales

Configuration de la hiérarchie mémoire

Différentes configurations de la hiérarchie mémoire sont utilisées dans les expérimentations présentées ci-après. Les paramètres, taille et associativité, des caches de données de premier et second niveau varient selon différentes configurations détaillées dans le tableau 2.2. La taille des lignes des caches est fixée à 32 octets.

Exception faite de la configuration *ideal*, la hiérarchie mémoire est composée de deux niveaux gérés selon une politique dite *non-inclusive* ou *mostly-inclusive*. Tous les caches implémentent une politique de remplacement *least recently used* (voir exemple 1.3, p. 15). Le cache d'instructions de premier niveau est supposé parfait ; le cache d'instructions ne connaît pas de défauts. Pour l'analyse d'architectures multi-cœurs, chaque cœur dispose de caches de premier niveau privés et seul le second niveau est partagé.

Les latences d'accès au premier et second niveau, et à la mémoire sont respectivement de 1, 5 et 15 cycles. L'utilisation d'une politique d'écriture de type *write-through & no-write-allocate*, sans tampon d'écriture, implique une latence de 15 cycles pour chaque écriture, le temps de propager la valeur écrite dans tous les niveaux de la hiérarchie.

Environnement d'analyse

Les expérimentations présentées ci-après ont été conduites sur des exécutables Mips R2000/R3000 produits, sans optimisation, par le compilateur gcc en version 4.5.2. L'édition de lien utilise le plan mémoire par défaut.

Le pire temps d'exécution (WCET, *worst case execution time*) des tâches analysées est calculé à l'aide du logiciel Heptane qui repose sur une méthode d'énumé-

TABLE 2.2 – Configurations des caches de données considérées dans les analyses du comportement temporel de la hiérarchie mémoire.

	Taille du cache		**Associativité**	
	L1	L2	L1	L2
tiny	32B	256B	1	8
small	256B	2KB	1	8
medium	1KB	4KB	1	8
ideal	128MB	-	1024	-

ration implicite des chemins (IPET). L'estimation de la contribution temporelle de chaque référence mémoire vis-à-vis des différents niveaux de la hiérarchie mémoire est obtenue en utilisant les méthodes introduites précédemment.

Avant l'analyse des contenus de cache, l'analyse d'adresses permet d'estimer la cible de chaque référence en mémoire. Si cette dernière ne peut être déterminée de façon exacte, l'analyse d'adresses approxime la plage d'adresses cible de la référence mémoire. Dans le cadre de cette étude, l'analyse utilisée est tirée de [38]. Elle permet d'obtenir la cible précise d'accès à des scalaires qu'ils soient globaux ou sur la pile. Pour les accès aux tableaux, notamment dans les boucles, l'intervalle d'adresses complet du tableau cible est retourné.

Afin d'étudier les effets des caches de données, l'estimation du WCET ne prend en compte que la contribution de ces derniers. Les effets d'autres mécanismes architecturaux ne sont pas considérés. En particulier, les anomalies temporelles [64] causées par des interactions entre pipeline et caches ne sont pas considérées. De fait, il est sûr de considérer qu'une référence mémoire non classifiée (NC) se comporte comme une référence toujours en défaut (AM).

Les latences d'accès aux différents niveaux de la hiérarchie sont bornées et connues. Dans le cas de caches partagés reliés aux différents cœurs d'exécution par un bus commun, cette borne peut être obtenue par l'utilisation d'un arbitre de bus approprié [74].

Codes considérés

Deux jeux de tâches sont utilisés dans le cadre de ces expérimentations. Le premier est composé d'un sous-ensemble des WCET *benchmarks*, codes maintenus par les groupes de recherche de l'université de Mälardalen [34]. Le second est issu d'une application réelle nommée debie [42] et développé par la société *Space Systems Finland Ltd* (SSF). Debie a pour objectif la mesure d'impacts de micro météorites et

de petits débris sur un satellite.

Les résultats fournis par la suite sont groupés par jeu de tâches, dénommés debie et malardalen, en fonction de la provenance respective des codes étudiés.

Le tableau 2.3 résume les caractéristiques de chacun des codes considérés, c'est à dire la taille des segments de code et de données (subdivisée en la taille du segment de données initialisées ou non, et celle de la pile).

TABLE 2.3 – Caractéristiques des différents segments (code, données initialisées ou non et pile) des tâches étudiées.

DEBIE

Tâche	**Code** (octets)	**Données initialisées** (octets)	**Données non initialisées** (octets)	**Pile** (octets)
acquisition_task	6168	5168	101904	161
hit_trigger_handler	2684	543	218	68
monitoring_task	12732	564	65772	250
tc_execution_task	13264	50	101444	164
tc_interrupt_handler	2980	31	35648	25
tm_interrupt_handler	972	23	168	20

MALARDALEN

Tâche	**Code** (octets)	**Données initialisées** (octets)	**Données non initialisées** (octets)	**Pile** (octets)
crc	1376	274	768	49
matmult	868	4	4800	24
minver	4540	88	432	124
nsichneu	44028	4	56	1516
sqrt	532	0	0	48
st	4896	5264	16000	264

2.4.2 Caches de données en environnement uni-cœur

Dans un premier temps, nous nous concentrons sur l'estimation de la précision des estimations temporelles dans un système uni-cœur en considérant un cache de données seul puis une hiérarchie mémoire comprenant deux niveaux de cache. Pour les résultats présentés dans cette section, les tâches sont donc considérées en isolation.

Précision de l'analyse d'un cache de données

Pour évaluer la précision intrinsèque de l'analyse d'un cache de données, nous estimons les défauts rencontrés par un cache arbitrairement grand en utilisant la configuration *ideal*. À l'exécution, dans ces conditions idéales, les tâches ne rencontrent que des défauts compulsifs liés au premier accès à chaque bloc de données. Le nombre de blocs de données manipulés par l'application est donc une borne supérieure du nombre de défauts qu'elle subit. La comparaison entre comportement estimé et simulé serait plus précise mais, à l'exception de simples noyaux de calcul, le pire chemin d'un code n'est pas forcément connu rendant la dite comparaison caduque.

Pour chaque code, son pire chemin d'exécution est calculé par l'analyseur Heptane en considérant la contribution temporelle d'une hiérarchie mémoire correspondant à la configuration *ideal*. Pour chaque jeu de tâches et chaque tâche (de haut en bas), le tableau 2.4 présente le nombre estimé de références mémoire (*Références*, colonne 2), de défauts de cache (*Défauts*, colonne 3) rencontrés sur son pire chemin et le nombre de blocs de cache différents accédés sur ce même chemin (*Blocs de données*, colonne 4). Une différence entre le nombre de défauts et le nombre de blocs accédés dans cette configuration implique une perte de précision du fait de la méconnaissance de la cible exacte d'une référence mémoire ou du chemin suivi au sein de l'application.

Dans certains cas, le pessimisme de l'analyse est restreint. La différence entre le nombre de blocs accédés et le nombre de défauts estimés est alors négligeable (*tc_execution_task*, *sqrt*, *matmult*) voir nulle (*tm_interrupt_handler*). Une telle estimation est possible pour des applications avec un flot de contrôle simple, c'est à dire présentant peu de convergence entre des chemins distincts (*tc_execution_task*) ou un unique chemin (*matmult*).

L'application *minver* illustre l'impact de l'indéterminisme d'accès sur la précision des analyses de cache. Le flot de contrôle de *minver* se compose d'un unique chemin. La tâche est une succession de courtes boucles, chacune traitant un ensemble de tableaux de données. Une référence mémoire visant un de ces tableaux cible une plage d'adresses couvrant plusieurs blocs. Les succès en cache sont capturés par l'analyse de persistance qui capture la localité temporelle au sein de chaque boucle sous la forme d'une classification FM. Toutefois, l'indéterminisme dans la cible d'une référence mémoire ne permet pas de garantir en sortie de la boucle la présence de ses blocs en cache. Les structures ciblées sont supposées rechargées dans les boucles suivantes.

TABLE 2.4 – Estimation du nombre de défauts pour un cache arbitrairement grand (configuration *ideal* : L1 128MB, 1024 voies) en relation avec le nombre de blocs de données manipulés par l'application.

DEBIE

Tâche	**Références**	**Défauts**	**Blocs de données**
	sur le pire chemin		
acquisition_task	18714	4547	1143
hit_trigger_handler	3374	94	27
monitoring_task	12612482	436	49
tc_execution_task	4649	271	269
tc_interrupt_handler	61	20	17
tm_interrupt_handler	18	8	8

MALARDALEN

Tâche	**Références**	**Défauts**	**Blocs de données**
	sur le pire chemin		
crc	27999	112	37
matmult	28929	154	153
minver	1067	164	20
nsichneu	5166	326	51
sqrt	352	8	5
st	104599	13506	676

Les résultats pour *acquisition_task* sont eux aussi soumis aux aléas de l'indéterminisme d'accès. La tâche enregistre un évènement dans une file stockée en mémoire sous la forme d'un tableau. Chaque exécution de la tâche manipule un évènement précis, une même case de la file des évènements. Toutefois, du point de vue de l'analyse d'adresses, chaque référence à cet évènement peut cibler l'intégralité de la file. Pour l'analyse de cache, l'approche sûre est de considérer que des références successives à ce même évènement ciblent en fait des portions différentes de la mémoire.

Dans le cadre de debie, *monitoring_task* subit le coût de l'indéterminisme de chemin. Du fait de ses nombreuses ramifications, la plupart des références mémoire de cette tâche sont conditionnées. En sortie de chemin conditionné, nulle garantie sur le contenu de cache ne peut donc être fournie quand aux structures présentes en cache ; lors de la convergence de plusieurs flots d'exécution, la fonction de conjonction *Join* ne peut conserver pour sûr en cache que les blocs présents sur tous les flots.

st constitue un cas intéressant. Cette tâche repose sur le passage de pointeurs en paramètre de ses sous-routines. Dans notre configuration expérimentale en l'absence d'une analyse de pointeurs [53] coûteuse par nature, la convention de l'analyse d'adresses est de supposer qu'une référence mémoire utilisant un pointeur peut cibler n'importe quel bloc de données utilisé par la tâche. Les évaluations confirment que cette perte de précision se répercute par la suite sur les analyses de contenu de cache.

Considération de la hiérarchie complète

Afin de montrer la précision de la prise en compte dans les analyses de cache, de la totalité de la hiérarchie mémoire, deux estimations de sa contribution au pire temps d'exécution sont comparées dans le tableau 2.5. Dans le premier cas (WCET_{L1}, ligne 1), seul le cache L1 est considéré et tous les accès au cache de second niveau sont définis comme des défauts. Dans le second cas ($\text{WCET}_{L1\&L2}$, ligne 2), le contenu des deux niveaux de cache est analysé et pris en compte. Si les deux valeurs WCET_{L1} et $\text{WCET}_{L1\&L2}$ sont identiques, l'analyse du cache de second niveau n'apporte aucun bénéfice. Toute différence implique un gain de précision lié à la prise en compte de ce second niveau. Plus la différence est importante, plus l'est le gain de précision.

Dans le tableau 2.5, WCET_{L1} et $\text{WCET}_{L1\&L2}$ sont présentés pour chacune des applications analysées (de haut en bas), et configurations de caches considérées

(*tiny*, *small* et *medium*, en colonnes 3, 4 et 5 respectivement).

Dans une majorité des cas, considérer le cache de second niveau permet un gain de précision de l'estimation de la contribution au WCET de la hiérarchie mémoire. Par exemple, l'amélioration de précision atteint environ 37% pour *sqrt* dans la configuration *tiny*. Pour d'autres tâches, considérer le cache de second niveau n'apporte aucune amélioration (*matmult* en configuration *small* ou *medium*) ou des améliorations marginales (moins de 5% pour *st*, *tc_execution_task* ou *acquisition_task* dans les plus grosses configurations). Toutefois, même dans les cas où l'amélioration relative n'est pas forcément importante (environ 3% pour *st* en configuration *medium*), le gain absolu peut l'être (environ 60000 cycles) et justifie l'utilisation d'une analyse de caches multi-niveau.

L'augmentation de la taille du cache de premier niveau permet une meilleure exploitation de la localité des tâches. Il est intéressant d'observer que cet incrément est capturé par les analyses. WCET_{L1} diminue d'environ 30% entre la configuration *tiny* et la configuration *small* pour *crc*. Similairement, les bénéfices obtenus par utilisation d'un cache de second niveau plus grand sont capturés par l'analyse, toutefois ces bénéfices sont moindres. De plus, à mesure que la taille du premier niveau augmente, la pression sur les niveaux inférieurs diminue ; $\text{WCET}_{L1\&L2}$ tend vers WCET_{L1} à mesure que croît le L1.

Un WCET plus précis, tel qu'obtenu en considérant le second niveau de cache, peut aussi s'avérer bénéfique lors du dimensionnement des ressources allouées à un système. Ce cas est illustré par *hit_trigger_handler* pour laquelle WCET_{L1} en configuration *small* est supérieur à $\text{WCET}_{L1\&L2}$ en configuration *tiny*. L'analyse du comportement du second niveau permet dans une configuration utilisant peu de ressources *(tiny)* l'obtention d'une estimation plus fine que dans le cadre d'une configuration plus gourmande *(small)* où seul le premier niveau serait considéré.

Les tâches comme *matmult* travaillent sur des tableaux et matrices ne tenant pas simultanément en cache. Si la localité temporelle entre les différentes itérations des boucles qui parcourent ces structures existe, elle n'est pas capturée. Dans le cadre des analyses effectuées, les accès aux différents blocs mémoire d'un tableau ne sont pas ordonnés dans le temps ; les itérations successives sont supposées accéder des blocs différents pour des raisons de sûreté.

Ces résultats montrent que la prise en compte des hiérarchies de caches de données dans le cadre de l'estimation du pire temps d'exécution permet des estimations plus précises. De plus, la considération de plusieurs niveaux de cache permet de mitiger l'impact de l'indéterminisme sur l'analyse des niveaux précédents. Dans tous les cas, les résultats obtenus par l'analyse de la hiérarchie dans

TABLE 2.5 – Estimation de la contribution, en cycles, au pire-temps d'exécution d'une hiérarchie de caches de données dans différentes configurations. Cache L1 à correspondance directe, L2 associatif par ensemble (8 voies). *tiny* : L1 32B/L2 256B, *small* : L1 256B/L2 2KB, *medium* : L1 1KB/L2 4KB.

DEBIE

Application	**Métrique**	**tiny**	**small**	**medium**
acquisition_task	WCET_{L1}	507229	479629	478389
	$\text{WCET}_{L1\&L2}$	373054	478609	477609
hit_trigger_handler	WCET_{L1}	107468	102785	80128
	$\text{WCET}_{L1\&L2}$	88958	73391	67209
monitoring_task	WCET_{L1}	385145056	384465824	289256224
	$\text{WCET}_{L1\&L2}$	330836928	286883904	247274432
tc_execution_task	WCET_{L1}	110550	84750	84590
	$\text{WCET}_{L1\&L2}$	79605	84705	84590
tc_interrupt_handler	WCET_{L1}	1767	1622	1432
	$\text{WCET}_{L1\&L2}$	1561	1472	1402
tm_interrupt_handler	WCET_{L1}	654	620	529
	$\text{WCET}_{L1\&L2}$	559	544	529

MALARDALEN

Application	**Métrique**	**tiny**	**small**	**medium**
crc	WCET_{L1}	560376	436216	414036
	$\text{WCET}_{L1\&L2}$	453501	414860	409090
matmult	WCET_{L1}	1133626	1101666	1101666
	$\text{WCET}_{L1\&L2}$	1109686	1101666	1101666
minver	WCET_{L1}	25368	18568	13368
	$\text{WCET}_{L1\&L2}$	22413	15103	13368
nsichneu	WCET_{L1}	124975	78415	72275
	$\text{WCET}_{L1\&L2}$	124975	71725	68285
sqrt	WCET_{L1}	8491	4291	4291
	$\text{WCET}_{L1\&L2}$	5401	4291	4291
st	WCET_{L1}	2296582	2030562	1950382
	$\text{WCET}_{L1\&L2}$	2129167	1910787	1890562

son intégralité sont au moins identiques, et souvent meilleurs, à ceux obtenus par en considérant uniquement le premier niveau.

2.4.3 Caches de données en environnement multi-cœur

Dans cette section, nous évaluons maintenant l'analyse de cache de données partagés et plus particulièrement l'impact des conflits. Dans un premier temps, aucune méthode de réduction des conflits n'est utilisée. Lors d'une seconde étape, l'impact du bypass sur les conflits et la précision des analyses de contenu de cache est évalué.

Sur la base des configurations *tiny*, *small* et *medium* (voir tableau 2.2 p. 46), le premier niveau de la hiérarchie mémoire est supposé privé à chaque cœur. Le second est partagé entre les différents cœurs d'exécution. Le comportement dans les trois configurations étant similaire, nous nous concentrons par la suite sur la configuration *medium*. Cette dernière dispose en effet des caches les plus importants et donc théoriquement les moins sensibles aux conflits inter-tâches. Afin d'autoriser la comparaison aux comportement estimés en contexte uni-cœur, les latences d'accès aux différents niveaux de la hiérarchie mémoire restent de 1, 5 et 15 cycles respectivement pour les deux premiers niveaux de cache et la mémoire principale.

Les tâches de chaque jeu, malardalen et debie, sont regroupées en deux systèmes composés chacun de 6 tâches concurrentes. malardalen comprend les tâches *crc*, *matmult*, *minver*, *nsichneu*, *sqrt* et *st*. debie est composé des tâches *acquisition_task*, *hit_trigger_handler*, *monitoring_task*, *tc_execution_task*, *tc_interrupt_handler* et *tm_interrupt_handler*. Les tâches d'un système ne partagent aucune donnée. De plus, en l'absence d'information sur la répartition des tâches sur les différents cœurs durant l'analyse, chaque tâche est supposée s'exécuter sur un cœur en concurrence avec les autres réparties sur un ou plusieurs autres cœurs.

Impact des conflits sur les analyses de cache partagés

Nous nous intéressons ici à l'évaluation du comportement d'un cache de données partagé en environnement multi-cœur sous la contrainte de conflits inter-tâches. Aucun mécanisme spécifique n'est mis en œuvre pour réduire ces conflits ou leur impact. Chaque tâche se retrouve en conflit avec les cinq autres tâches du système.

Les tableaux 2.6 et 2.8 présentent les résultats de cette étude respectivement pour un système composé des tâches debie et malardalen. Pour chaque tâche d'un système sont donnés de haut en bas : son pire temps d'exécution en considérant les conflits sur le cache partagé de second niveau ($\text{WCET}_{L1\&L2}$, ligne 1), et le nombre

de défauts subis au premier et au second niveau de la hiérarchie mémoire (*Défauts L1* et *Défauts L2* respectivement en lignes 2 et 3) sur le pire chemin estimé. Si ces deux valeurs sont identiques alors aucun succès ne peut être garanti de façon sûre au niveau du cache partagé. Les tableaux 2.7 et 2.9 présentent la contribution des tâches de debie et malardalen aux conflits subis par leurs rivales. Chaque entrée correspond au nombre de lignes de cache considérées comme occupées par une tâche lorsqu'elle est en concurrence avec la tâche analysée. Le cache de second niveau considéré peut héberger 128 blocs différents.

Dans le cas de debie comme de malardalen, quelle que soit la tâche analysée, les conflits générés par ses tâches concurrentes sont trop importants pour pouvoir garantir pour sûr le moindre succès dans le cache partagé. Dans les configurations étudiées, aucun des défauts du premier niveau ne peut être estimé comme capturé par le second niveau. Pour des raisons de sûreté, dans le cas de debie et de malardalen, 7603 et respectivement 942 lignes de cache différentes sont considérées comme occupées durant l'analyse du cache L2 qui n'en peut contenir que 128.

Sans mécanisme pour contrôler les conflits inter-tâches, une tâche comme *acquisition_task*, à elle seule, est considérée occuper 3197 lignes de cache différentes, soit environ 100Ko. Il faudrait pour qu'une tâche concurrente puisse commencer à bénéficier du cache de façon sûre que ce dernier dépasse les 100Ko de taille.

Le cas de *tc_interrupt_handler* permet d'évaluer l'impact des abstractions d'ordre et d'occurrence des accès sur l'estimation des conflits qu'elle génère. Si sur un nombre suffisamment grand d'exécutions différentes la tâche peut manipuler 1123 blocs différents, sur son pire chemin d'exécution elle n'en manipule que 17. Il serait donc intéressant pour chaque tâche rivale d'estimer le nombre maximum de blocs qu'elle peut manipuler lors d'une instance. L'estimation ainsi obtenue, en combinaison avec sa fréquence relative par rapport à celle de la tâche analysée, pourrait permettre d'obtenir des estimations plus fines des conflits qu'elle génère.

En résumé, à moins de ne considérer que de petits systèmes ou des caches suffisamment importants, l'analyse de caches de données partagés conduit à des résultats pessimistes. Sans contrôle des conflits inter-tâches, nulle tâche ne peut bénéficier de façon sûre des caches partagés. Le problème, déjà présent dans le contexte des caches d'instructions [37], est exacerbé pour les caches de données où le volume mémoire manipulé tend à être plus important.

TABLE 2.6 – Estimation de l'impact des conflits inter-tâches sur le comportement estimé d'une hiérarchie de caches de données pour le jeu de tâches concurrentes debie. Cache L1 privé à correspondance directe, L2 partagé associatif par ensemble (8 voies). Configuration *medium* : L1 1KB/L2 4KB.

DEBIE

Tâche	**Métrique**	**Configuration medium** (avec conflits)
acquisition_task	$\text{WCET}_{L1\&L2}$	478389
	Défauts L1	11541
	Défauts L2	11541
hit_trigger_handler	$\text{WCET}_{L1\&L2}$	80128
	Défauts L1	971
	Défauts L2	971
monitoring_task	$\text{WCET}_{L1\&L2}$	289256224
	Défauts L1	3597804
	Défauts L2	3597804
tc_execution_task	$\text{WCET}_{L1\&L2}$	84590
	Défauts L1	1292
	Défauts L2	1292
tc_interrupt_handler	$\text{WCET}_{L1\&L2}$	1432
	Défauts L1	23
	Défauts L2	23
tm_interrupt_handler	$\text{WCET}_{L1\&L2}$	529
	Défauts L1	8
	Défauts L2	8

TABLE 2.7 – Contribution aux conflits sur le cache partagé des tâches de debie. Configuration *medium* : L1 privé 1KB/L2 partagé 4KB.

DEBIE

Tâche	**Conflits générés** (blocs de cache)
acquisition_task	3197
hit_trigger_handler	33
monitoring_task	50
tc_execution_task	3187
tc_interrupt_handler	1123
tm_interrupt_handler	13

TABLE 2.8 – Estimation de l'impact des conflits inter-tâches sur le comportement estimé d'une hiérarchie de caches de données pour le jeu de tâches concurrentes malardalen. Cache L1 privé à correspondance directe, L2 partagé associatif par ensemble (8 voies). Configuration *medium* : L1 1KB/L2 4KB.

MALARDALEN

Tâche	**Métrique**	**Configuration medium** (avec conflits)
crc	$\text{WCET}_{L1\&L2}$	414036
	Défauts L1	446
	Défauts L2	446
matmult	$\text{WCET}_{L1\&L2}$	1101666
	Défauts L1	24005
	Défauts L2	24005
minver	$\text{WCET}_{L1\&L2}$	13368
	Défauts L1	166
	Défauts L2	166
nsichneu	$\text{WCET}_{L1\&L2}$	72275
	Défauts L1	632
	Défauts L2	632
sqrt	$\text{WCET}_{L1\&L2}$	4291
	Défauts L1	8
	Défauts L2	8
st	$\text{WCET}_{L1\&L2}$	1950382
	Défauts L1	48171
	Défauts L2	48171

TABLE 2.9 – Contribution aux conflits sur le cache partagé des tâches de malardalen. Configuration *medium* : L1 privé 1KB/L2 partagé 4KB.

MALARDALEN

Tâche	**Conflits générés** (blocs de cache)
crc	37
matmult	153
minver	20
nsichneu	51
sqrt	5
st	676

Application du bypass pour restreindre les conflits inter-tâches

Le bypass permet de contrôler la contribution d'une instruction aux conflits générés par sa tâche. Dans cette section, nous évaluons cette capacité à réduire les interférences et à améliorer la précision des analyses de caches partagées prenant en compte les conflits.

Les effets des différentes heuristiques proposées, CB,AB et IB, sont présentés dans les tableaux 2.10 et 2.12 pour les jeux de tâches debie et malardalen respectivement. Lorsqu'une heuristique est appliquée, la même heuristique est utilisée pour toutes les tâches du système. Seul le cache de second niveau, partagé entre les différents cœurs du système, est affecté par le bypass. Pour chaque tâche d'un système sont donnés de haut en bas : son pire temps d'exécution ($\text{WCET}_{L1\&L2}$, ligne 1), le nombre de défauts subis au premier et au second niveau de la hiérarchie mémoire (*Défauts L1* et *Défauts L2* respectivement en lignes 2 et 3) sur le pire chemin estimé par l'analyseur. Si un nombre égal de défauts est estimé aux deux niveaux, cela montre que l'on ne peut bénéficier du L2 du point de vue de l'analyse. Les tableaux 2.11 et 2.13 présentent la contribution des tâches de debie et malardalen aux conflits subis par leurs rivales sous les heuristiques de bypass proposées. Pour chaque tâche, cela correspond donc au nombre de lignes de cache qu'elle est considérée occuper lorsqu'elle se retrouve en concurrence avec une tâche analysée.

Pour chacun des systèmes considérés, au moins l'une des heuristiques de bypass permet de réduire les conflits de façon suffisamment significative pour permettre la capture de succès au niveau du cache partagé. CB bénéficie au tâches de debie ; AB et IB bénéficient à la fois aux tâches de debie et à celles de malardalen. Sans bypass, comme étudié dans la section précédente, aucun succès au niveau du cache partagé ne pouvait être capturé de façon sûre. Par rapport à un système où les conflits ne sont pas restreints, le constat est positif. Les estimations de pire temps d'exécution obtenues sont plus précises et, au pire, identiques.

Certaines tâches, comme *matmult* ou *minver*, ne bénéficient pas du point de vue des analyses du cache de second niveau sous la configuration *medium*. Ce comportement est capturé par les heuristiques et sous bypass elles voient leur contribution aux conflits diminuer pour n'atteindre qu'une poignée de blocs. Les conflits générés par *matmult* se limitent à 3 blocs au maximum en utilisant les heuristiques proposées contre 153 sans restriction des conflits.

Pour malardalen, l'heuristique CB est efficace sur la majorité des tâches mais s'avère insuffisante. La contribution au conflits de *st* (672 blocs) suffit à bloquer le

cache (128 blocs) pour ses rivales. Les heuristiques plus agressives (AB) ou ciblées vers les accès non déterministes (IB) diminuent sa contribution jusqu'à 0 ou 11 blocs respectivement. Dans le cas AB, aucun des 323 accès de la tâche n'est autorisé à insérer de blocs de données dans le cache partagé.

Dans le cas de debie et quelle que soit l'heuristique utilisée, la réduction des interférences générées par chaque tâche reste forte. Sans restriction, la contribution au conflits de *acquisition_task* et *tc_execution_task* se chiffre en milliers de blocs de cache. En utilisant le mécanisme de bypass, seule une dizaine de blocs utiles est conservée pour chacune, blocs dont la localité est capturée dans le cache de second niveau.

Le bypass permet aussi de réduire les conflits intra-tâches ainsi qu'illustré par l'heuristique IB sur la tâche *st*. Son pire temps estimé en utilisant un cache partagé, 1590112 cycles, est inférieur à son pire temps estimé en utilisant la même hiérarchie mémoire sans bypass dans un contexte uni-cœur, 1890562 cycles. Les accès non déterministes peuvent en effet en masquer d'autres plus prédictibles et provoquer l'éviction de données utiles.

On constate de par ces résultats que le bypass est un mécanisme viable pour diminuer la pression sur les caches partagés sans les sous-exploiter. Les heuristiques conservatives ciblent les accès dont les bénéfices ne sont pas capturés par les analyses. Dans le cas de tâches peu sujettes à l'indéterminisme, qu'il soit d'accès ou de chemin, des politiques plus agressives sont requises pour amener la pression sur les caches partagés à un niveau raisonnable. De fait, à mesure que la précision des analyses de contenu de cache s'améliorera, les bénéfices des politiques conservatrices s'amenuiseront.

TABLE 2.10 – Estimation de l'impact des heuristiques de bypass CB (conservative), AB (agressive) et IB (non déterministe) sur les conflits inter-tâches en environnement multi-cœur. Jeu de tâche debie sur cache L1 privé à correspondance directe, L2 partagé, sous bypass, par ensemble (8 voies). Configuration *medium* : L1 1KB/L2 4KB.

DEBIE

Tâche	**Métrique**	**Configuration medium avec conflits**		
		CB	AB	IB
acquisition_task	WCET$_{L1\&L2}$	477594	478374	364179
	Défauts L1	11541	11541	11541
	Défauts L2	11488	11540	3927
hit_trigger_handler	WCET$_{L1\&L2}$	67224	75598	71188
	Défauts L1	970	971	971
	Défauts L2	112	669	375
monitoring_task	WCET$_{L1\&L2}$	235296672	262276304	253400272
	Défauts L1	3597800	3597800	3597804
	Défauts L2	508	1799148	1207408
tc_execution_task	WCET$_{L1\&L2}$	84590	84590	73055
	Défauts L1	1292	1292	1292
	Défauts L2	1292	1292	523
tc_interrupt_handler	WCET$_{L1\&L2}$	1402	1402	1402
	Défauts L1	23	23	23
	Défauts L2	21	21	21
tm_interrupt_handler	WCET$_{L1\&L2}$	529	529	529
	Défauts L1	8	8	8
	Défauts L2	8	8	8

TABLE 2.11 – Estimation de l'impact des heuristiques de bypass CB (conservative), AB (agressive) et IB (non déterministe) sur la contribution aux conflits du jeu de tâche debie. Configuration *medium* : L1 privé 1KB/L2 partagé 4KB.

DEBIE

Tâche	**Conflits générés (blocs)**		
	CB	AB	IB
acquisition_task	15	1	16
hit_trigger_handler	23	5	9
monitoring_task	47	25	20
tc_execution_task	13	9	16
tc_interrupt_handler	1	1	7
tm_interrupt_handler	0	0	6

TABLE 2.12 – Estimation de l'impact des heuristiques de bypass CB (conservative), AB (agressive) et IB (non déterministe) sur les conflits inter-tâches en environnement multi-cœur. Jeu de tâche malardalen sur cache L1 privé à correspondance directe, L2 partagé, sous bypass, par ensemble (8 voies). Configuration *medium* : L1 1KB/L2 4KB.

MALARDALEN

Tâche	**Métrique**	**Configuration medium avec conflits**		
		CB	AB	IB
crc	$\text{WCET}_{L1\&L2}$	414036	410800	411501
	Défauts L1	446	444	446
	Défauts L2	446	232	277
matmult	$\text{WCET}_{L1\&L2}$	1101666	1101666	1101666
	Défauts L1	24005	24005	24005
	Défauts L2	24005	24005	24005
minver	$\text{WCET}_{L1\&L2}$	13368	13368	13368
	Défauts L1	166	166	166
	Défauts L2	166	166	166
nsichneu	$\text{WCET}_{L1\&L2}$	72275	68285	68285
	Défauts L1	632	632	632
	Défauts L2	632	366	366
sqrt	$\text{WCET}_{L1\&L2}$	4291	4291	4291
	Défauts L1	8	8	8
	Défauts L2	8	8	8
st	$\text{WCET}_{L1\&L2}$	1950382	1950382	1590112
	Défauts L1	48171	48171	48171
	Défauts L2	48171	48171	24153

TABLE 2.13 – Estimation de l'impact des heuristiques de bypass CB (conservative), AB (agressive) et IB (non déterministe) sur la contribution aux conflits du jeu de tâche malardalen. Configuration *medium* : L1 privé 1KB/L2 partagé 4KB.

MALARDALEN

Tâche	**Conflits générés (blocs)**		
	CB	AB	IB
crc	37	12	4
matmult	2	1	3
minver	14	14	5
nsichneu	51	51	50
sqrt	2	1	3
st	672	0	11

Conclusion

Ce chapitre a présenté des méthodes permettant la prise en compte de la contribution temporelle de caches de données au sein d'une hiérarchie mémoire. Les analyses proposées ont ensuite été étendues afin de prendre en compte les conflits inter-tâches. Ces conflits apparaissent dans le cadre d'architectures multi-cœurs au niveau des caches de données partagés. Dans l'optique de réduire l'impact des interférences, nous avons proposé différentes heuristiques basées sur le mécanisme de bypass, mécanisme dont les compétences ont déjà été démontrées dans le cadre de caches d'instructions.

Les résultats expérimentaux montrent le gain de précision apporté par la considération de la hiérarchie complète sur différentes configurations en environnement uni-cœur. Dans le cadre de caches partagés, les conflits même sur des systèmes modestes tendent à handicaper les analyses. Pour des raisons de sûreté, le cache se comporte comme si il était intégralement occupé par les blocs de tâches concurrentes. Les heuristiques de bypass proposées s'avèrent des outils intéressants afin de relâcher la pression exercée par un ensemble de tâches sur le cache partagé quand cette dernière n'est pas exagérée.

Perspectives

Exploration du bypass. Les heuristiques de bypass proposées reposent sur la prévention de l'insertion de blocs en cache. Les blocs candidats sont ceux dont la réutilisation ne peut être garantie par les analyses de cache. À mesure que les analyses de cache vont gagner en précision, le nombre de références candidates au bypass risque de péricliter laissant les caches partagés en proie aux conflits.

Au lieu d'heuristiques considérant les références mémoire en isolation et visant à évaluer la contribution de chacune, il serait intéressant d'explorer des heuristiques ayant une vue plus globale d'une tâche, d'un groupe de tâches ou du système. L'objectif reste de réduire la pression sur les ressources partagées jusqu'à atteindre un niveau raisonnable. L'utilisation des ressources disponibles toutefois ne doit pas être trop faible sous peine de les sous-utiliser.

En considérant une tâche en isolation, le bypass dispose aussi d'avantages soulevés dans nos expérimentations pour le cas de *st*. Une utilisation judicieuse du bypass permet de réduire les conflits intra-tâches. Par exemple, il est possible de sélectionner pour le bypass les accès dont la contribution n'est pas capturée et qui masquent d'autres références plus utiles. L'étude de ce comportement et la définition d'heuristiques appropriées permettrait de réduire le pire temps d'exécution

de tâches critiques et faciliter la validation de systèmes temps-réel.

Un autre aspect non étudié du bypass et des heuristiques proposées dans cet étude est leur impact sur les performances moyennes des applications qui y sont soumises. Si des heuristiques conservatives n'ont pas d'impact négatif sur le pire temps d'exécution estimé, les blocs non insérés en cache peuvent participer à une localité non capturée par les analyses de cache. De fait, l'utilisation du bypass dans les premiers niveaux de la hiérarchie mémoire est déconseillée.

Modélisation des caches unifiés. Les caches unifiés, contenant instructions et données, sont peu représentés dans les travaux existants. À notre connaissance, une seule étude [16] s'est préoccupée de la modélisation des caches unifiés pour les analyses de contenu. La fonction de transfert utilisée suppose le chargement d'une instruction directement suivi des données accédées par cette dernière. Cette modélisation simple s'avère correcte dans le cas d'une hiérarchie mémoire considérée en isolation de tout autre mécanisme architectural.

L'adjonction d'un pipeline basique séparant les étages de chargement d'une instruction, *instruction fetch*, et d'exécution de l'opération mémoire, *execute*, invalide ce modèle. En effet, entre le chargement d'une instruction et celui des données qu'elle manipule, d'autres instructions entrent dans le pipeline en *instruction fetch*. Les modifications induites ne sont pas prises en compte par la modélisation actuelle, faussant ainsi le calcul des classifications comportementales.

La prise en compte correcte de la contribution des caches unifiés au pire temps d'exécution dans un cadre général reste donc un problème ouvert. Le principal problème en la présente est l'estimation du délai entre les différents étages du pipeline considéré afin de pouvoir estimer la distance séparant le chargement d'une instruction du chargement de ses opérandes mémoire. Si ce délai ne peut être estimé de façon précise, les dates de disponibilité au plus tôt et au plus tard de l'opérande mémoire en cache doivent être estimées. La date au plus tard permet de garantir des succès lors d'accès consécutifs à cette donnée. La date au plus tôt permet d'évaluer l'impact d'autres accès sur la position logique de la donnée en cache et donc sa position au pire cas.

Modèle d'analyse de caches de données. Le modèle actuel d'analyse de contenu pour les caches de données repose sur une extension de celui proposé pour les caches d'instructions. La plupart des extensions proposées dans les travaux récents [7, 43] étendent la définition du contexte d'une instruction au-delà de son simple contexte d'appel. Pour des contextes plus précis, l'identification des réfé-

rences mémoire est raffinée et des plages mémoire cibles plus fines sont dérivées. Dans une mesure limitée, ces approches permettent aussi une prise en compte de l'ordonnancement temporel entre les différentes itérations d'une boucle.

Ces extensions toutefois ont un coût mal évalué dans les travaux récents en terme de complexité mémoire et de complexité de calcul. Combiné à une analyse de type interprétation abstraite, chaque raffinement contextuel augmente significativement le coût des analyses. Dans des travaux récents focalisés sur les caches d'instructions [36], nous avons montré que le remplacement du point fixe par des analyses ad hoc permettaient d'en réduire le coût avec une perte de précision négligeable.

L'étude des comportements capturés mais aussi non prédits dans le cadre des travaux actuels serait une base intéressante pour la définition de passes d'analyse ad hoc pour les caches de données. Par exemple, une telle analyse [35], basée sur le calcul de relations entre accès mémoire, a été étudiée dans une contribution récente.

Une autre piste à explorer pour améliorer la précision des analyses de contenu de cache est l'utilisation de transformations logiques du graphe de flot de contrôle de la tâche considérée. Le déroulage de la première itération de chaque boucle est un exemple de ces transformations. La duplication des blocs de base succédant un point de convergence en est une autre permettant de retarder la perte d'information après l'exécution d'un chemin conditionné.

Chapitre 3

PRETI : Caches partitionnés pour environnements temps-réel

La prise en compte des conflits inter-tâches peut nuire à la précision des analyses de caches partagés lorsque les tâches concurrentes manipulent d'importants volumes de données. Cette perte de précision se répercute sur les estimations temporelles qui en dérivent. Dans les cas les plus extrêmes, pour des raisons de sûreté, l'intégralité du cache est considérée comme occupée par les blocs de tâches concurrentes. Les travaux existants [58, 37, 56] s'accordent sur le besoin d'affiner ces estimations, ou de mettre en œuvre des méthodes afin de réduire la pression sur le cache. Toutefois, chaque nouvelle tâche qui peut s'exécuter en parallèle avec les tâches critiques rajoute à cette pression.

Le partitionnement est un mécanisme permettant d'assurer à une tâche, ou un groupe de tâches, la jouissance exclusive d'un volume de cache connu nommé *partition de cache*. Dans le cadre des systèmes multi-tâches, le partitionnement permet de prévenir les conflits inter-tâches liés à des préemptions ou à des accès concurrents ; dans un cadre plus général, le partitionnement permet l'isolation des performances de tâches critiques du point de vue de l'analyse de cache.

En contexte temps-réel, les mécanismes de partitionnement existants dans la littérature reposent sur une division stricte du cache, figée pour la durée de vie du système. Cette division est stricte en ce que chaque tâche est restreinte à sa partition, et l'espace d'une partition est dédié à son seul propriétaire. Une partition surdimensionnée implique donc une perte d'espace utile dans le cache. À l'inverse, si nulle provision n'est faite pour les tâches les moins critiques, elles se retrouvent sans espace de cache disponible. Ces restrictions sont en partie liées à l'implémentation du mécanisme de partitionnement. Si elles répondent aux contraintes de

prédictibilité requises pour les tâches critiques, c'est au sacrifice des performances des autres tâches.

Contributions

Dans ce chapitre, nous proposons la politique de partitionnement PRETI, *partitionned real-time* (*partitionned real-time*) pour caches partagés en environnement temps-réel [57]. En tant que mécanisme de partitionnement, PRETI permet l'isolation, dans des limites connues, des portions de caches utilisées par différentes tâches concurrentes. Le comportement des tâches critiques peut donc être estimé, pour chacune, par le biais des méthodes d'analyses classiques pour caches privés, en isolation de la connaissance de ses concurrentes.

De plus, PRETI est conçu pour bénéficier aux systèmes hybrides, combinant sur la même architecture tâches critiques et non critiques. L'espace non alloué, ou non utilisé, est accessible à tous les utilisateurs du cache afin de permettre de meilleures performances.

Dans les faits, le mécanisme est implémenté par une simple modification d'une politique de remplacement existante, la politique LRU (*least recently used*) favorisée pour sa prédictibilité. Son intégration dans les architectures existantes se fait à coût moindre.

Le mécanisme PRETI bénéficie d'un cadre d'application très large et convient à tous les systèmes multi-tâches, multi-cœurs ou à base de *simultaneous multi-threading*. Le mécanisme supporte l'allocation de partitions à un cœur, un *thread* ou une tâche. Notre présentation par la suite se focalisera toutefois sur une utilisation en contexte multi-cœur avec allocation de partitions à la granularité de la tâche.

Organisation

Ce chapitre s'ouvre sur la présentation des bases du mécanisme de partitionnement PRETI (§ 3.1), de la division logique du cache à la considération de ce dernier dans les analyses de cache. L'intérêt du mécanisme pour la prédictibilité des tâches critiques et les performances des tâches non critiques est ensuite évalué (§ 3.2). Nous présentons enfin des extensions du mécanisme pour améliorer sa polyvalence (§ 3.3), tout en conservant ses propriétés inhérentes.

3.1 Fondements de la politique de partitionnement PRETI

La politique de partitionnement PRETI repose sur une division par voies des niveaux de caches partagés. Étant donnée une tâche à laquelle est allouée N voies, PRETI garantit que ses N blocs les plus récemment utilisés sont conservés dans chaque ensemble du cache. Cette garantie assure la prédictibilité du mécanisme. Le mécanisme vise aussi à l'utilisation de l'espace du cache observé comme non alloué ou non utilisé par son propriétaire, notamment pour bénéficier aux tâches non critiques. Au contraire, des mécanismes plus stricts réservent exactement N blocs dans chaque ensemble du cache. Ces blocs sont dédiés à l'usage exclusif de leur propriétaire qui ne peut en utiliser plus en cache.

Par la suite, nous introduisons dans un premier temps les fondements logiques sur lesquels repose la politique de remplacement PRETI (§ 3.1.1). Sur ces bases, une implémentation possible (§ 3.1.2) et les analyses statiques (§ 3.1.3) pour des caches PRETI sont ensuite présentées.

3.1.1 Politiques d'accès et de mise à jour du cache

Espaces privés et espace partagé

Le mécanisme de partitionnement PRETI repose sur une division *logique* du cache en espaces distincts, distingués en espaces privés et un espace partagé tels qu'illustrés dans l'exemple 3.6. La modification de chacun, privé ou partagé, est restreinte en fonction de la tâche effectuant une requête sur le cache. En particulier, les politiques d'éviction, d'insertion et de promotion ne peuvent altérer l'ordre de blocs d'espaces privés concurrents. La lecture ou l'écriture du contenu d'un bloc présent dans le cache ne sont pas impactées.

Les **espaces privés** représentent les partitions individuelles de chaque tâche en cache. À un instant donné, chaque espace privé est alloué à une tâche unique et son contenu est modifié exclusivement par son propriétaire. Un espace privé maintient les blocs les plus récents du point de vue de la politique de remplacement de son propriétaire. Il s'agit d'une portion logique, délimitée par la politique de remplacement, et non physique de l'espace du cache. Le nombre de blocs maintenus, exprimé en nombre de voies par ensemble du cache, dépend de la taille de la partition allouée à la tâche. Les espaces privés permettent donc d'assurer l'application d'un partitionnement précalculé [84, 75, 10].

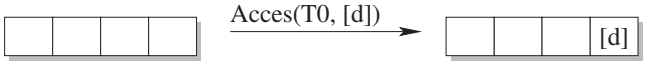
L'**espace partagé** est global et peut être utilisé pour le stockage des blocs de toute tâche du système, qu'elle dispose ou non d'un espace privé. Il est construit à la volée à partir des lignes de cache hébergeant des blocs en dehors de tout espace privé. L'espace partagé regroupe donc les blocs les moins récemment utilisés des tâches au delà de leur espace privé, s'il en est.

Un espace privé croît au fur et à mesure des besoins et des allocations en cache de son propriétaire, jusqu'à éventuellement atteindre sa capacité maximale. À contrario, l'espace partagé rétrécit à mesure que les tâches revendiquent effectivement l'espace privé qui leur est réservé. Une tâche, tant qu'elle n'utilise pas la totalité de sa partition, laisse donc des lignes libres qui sont incorporées au sein de l'espace partagé et donc utilisables par toutes les tâches du système.

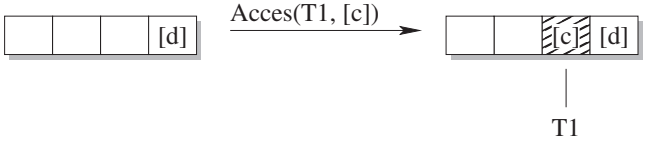
Exemple 3.6 – Division de l'espace d'un cache PRETI

Considérons un cache d'associativité 4. Les tâches T0, T1 et T2 se voient réserver respectivement 0, 1 et 2 voies dans le cache. Pour des raisons de lisibilité, seuls les accès sur un unique ensemble sont étudiés. Les lignes de cet ensemble sont présentées sans ordre logique particulier et font initialement partie de l'espace partagé :

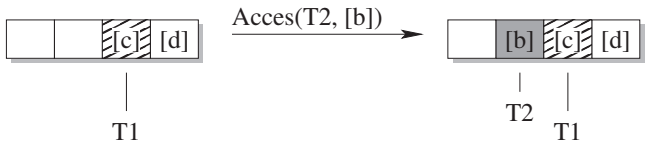
La tâche T0 n'ayant pas de partition dédiée, lors d'une insertion son bloc cible est inséré en espace partagé. Les 4 lignes de l'ensemble considéré restent partie de l'espace partagé :



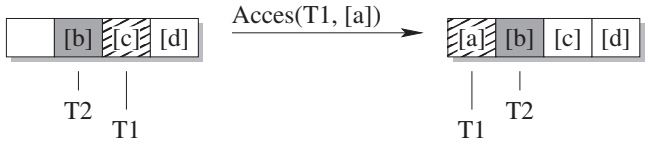
Lorsque la tâche T1 insère le bloc [c] en cache, elle réquisitionne une ligne au profit de son espace privé. Ce dernier arrive donc à capacité et l'espace partagé se retrouve réduit d'autant :



Lorsque la tâche T2 accède au cache, son espace privé croît de façon similaire, aux dépens de l'espace partagé :



Un nouvel accès de T1 au cache n'implique pas la croissance de son espace privé. Ce dernier est arrivé à capacité. Seul le bloc [a], le plus récemment utilisé est garanti conservé en espace privé :



L'espace partagé comprend deux lignes de cache, une ligne non allouée à une quelconque tâche et une ligne non encore réquisitionnée par son propriétaire. Les espaces privés de T1 et T2 conservent les blocs les plus récemment utilisés de leurs tâches respectives. Cette garantie s'étend à un et deux blocs respectivement pour T1 et T2. Pour T2, cela s'applique au seul bloc auquel elle accède. Pour T1, au bloc [a], [c] se retrouve en espace partagé d'où toute tâche peut l'évincer.

Politiques d'éviction, promotion et insertion des blocs

En cas de défaut de cache, il peut être nécessaire de trouver un bloc à évincer, dans l'ensemble cible, afin de libérer une ligne pour y insérer le bloc manquant. Cette sélection se base sur une organisation logique des blocs au sein de chaque ensemble. La politique mise en œuvre dans un cache PRETI dérive de la politique LRU, favorisée pour sa prédictibilité [81]. Les blocs sont organisés du plus récent, MRU (*Most Recently Used*), au plus ancien, LRU (*Least Recently Used*). C'est ce dernier qui est sélectionné par la politique d'éviction. Dans le cadre de PRETI, les modifications apportées à la politique de remplacement visent à assurer le maintien de l'exclusivité des modifications d'un espace privé à son seul propriétaire.

La **politique d'insertion** mise en œuvre par un cache PRETI est la même que celle utilisée dans un cache LRU classique. Une ligne nouvellement insérée en cache se retrouve marquée comme étant la plus récemment utilisée, MRU. Logiquement, le bloc entre dans l'espace privé de la tâche en ayant fait la requête. Pour conserver cet espace privé dans les limites de son allocation, son plus ancien bloc avant insertion peut être démis en espace partagé.

De même, la **politique de promotion** de PRETI est similaire à celle d'un cache LRU classique ; le bloc accédé se retrouve marqué comme étant le plus récemment utilisé. L'entrée et la sortie de blocs de l'espace privé interviennent comme dans le cadre de la politique d'insertion.

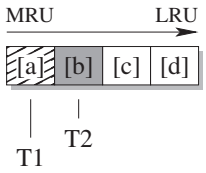
La **politique d'éviction** est le point de variation fondamental entre un cache PRETI et un cache LRU. Ces modifications visent à assurer le maintient de l'exclusivité des modifications d'un espace privé à son seul propriétaire. Ainsi, lors d'un défaut de cache, le bloc sélectionné pour éviction est le plus ancien qui répond à l'un de ces critères :

- appartient à l'espace partagé, si ce dernier n'est pas nul ;
- appartient à l'espace privé de la tâche provoquant le défaut, si ce dernier a atteint sa capacité.

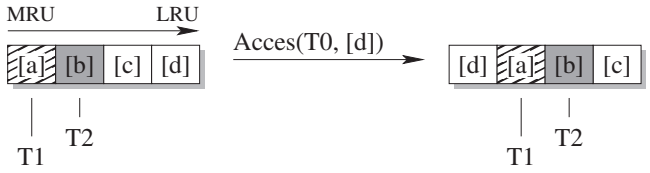
À titre de comparaison, dans le cadre d'une politique LRU classique, le bloc évincé serait celui qui est globalement le plus ancien, le moins récemment utilisé, de l'ensemble de cache cible. Ce qui appliqué dans un cache PRETI impliquerait l'éventuelle sélection d'un bloc de l'espace privé d'une tâche concurrente d'où une violation des tailles de partition réservées. Dans l'implémentation PRETI, l'espace partagé ou l'espace privé de la tâche faisant la requête priment sur les autres.

EXEMPLE 3.7 – Comportement d'un cache PRETI

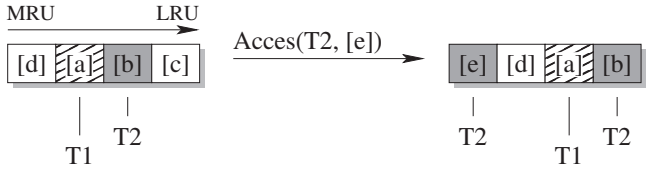
Nous nous plaçons dans la même configuration que dans l'exemple 3.6. Les tâches concurrentes du système T0, T1 et T2 se sont vues réserver respectivement 0, 1 et 2 voies dans le cache. Nous considérons toujours un cache d'associativité 4 dont nous ne représentons qu'un seul ensemble. Les lignes de celui-ci sont présentées dans leur ordre logique du plus récemment utilisé (MRU) au moins récemment utilisé (LRU) :



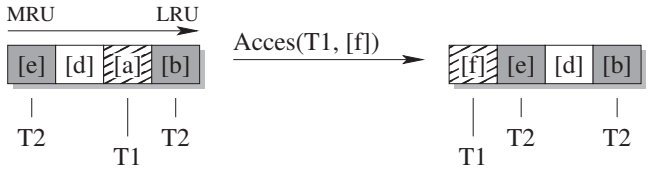
Pour la tâche T0 qui n'a pas de partition dédiée, insertion et promotion de blocs en cache se déroulent au sein de l'espace partagé. Comme illustré, lorsqu'elle accède au bloc [d], il est promu en position MRU dans l'espace partagé :



Avec l'accès au bloc [e], absent du cache, l'espace privé de la tâche T2 arrive à capacité. Le bloc est inséré en position MRU dans l'espace privé de la tâche. Le bloc le plus ancien [c] appartenant à l'espace partagé, il est évincé pour laisser la place à [e] :



L'accès par la tâche T1 au bloc [f] résulte en son insertion dans l'espace privé de T1 en position MRU. Le bloc le plus ancien [b] ne peut être évincé car il fait partie de l'espace privé de T2. Le pénultième bloc, [a], appartient à l'espace privé de T1 qui a atteint sa capacité. [a] est donc sélectionné pour éviction :



3.1.2 Implémentation du partitionnement

Les différences fondamentales liées à l'implémentation d'un cache PRETI, par rapport à un cache LRU classique, tiennent à la politique de remplacement. Lors d'un succès ou d'une insertion, les structures régissant l'ordre des blocs d'un ensemble subissent les mêmes modifications que dans le cadre d'un LRU classique. La latence d'accès au cache en cas de succès n'est donc pas impactée. Les modifications pour implémenter la politique de remplacement PRETI ciblent principalement la politique d'éviction et la gestion des entrées et sorties de blocs en espace privé.

Contrairement à un cache LRU classique, la politique d'éviction ne sélectionne pas le bloc le moins récemment utilisé du point de vue global. Le bloc évincé est le moins récemment utilisé d'un point de vue local, sur un sous-ensemble des lignes du cache, celles de l'espace partagé, s'il en est, ou de l'espace privé de la tâche effectuant la requête. Cette sélection requiert la possibilité de pouvoir discriminer les lignes appartenant à chaque espace privé et celles de l'espace partagé.

À cette fin, le propriétaire de chaque bloc en cache doit donc pouvoir être identifié. Un identifiant de tâche (*task id*) est donc attaché à chaque ligne de cache. Des tâches avec des espaces privés dédiés utilisent donc des identifiants distincts. Les autres tâches, moins ou non critiques, n'ayant accès qu'à l'espace partagé, elles peuvent partager un même identifiant *NULL*. De plus, afin d'identifier les lignes en sur allocation, pour chaque identifiant, le cache maintient la taille maximale de sa partition, en nombre de voies.

Du point de vue de l'espace de stockage requis, le coût d'une telle implémentation est relativement bas. Pour le stockage des identifiants de tâche $\lceil log_2(P+1) \rceil * \ell$ bits sont requis au total, où P désigne le nombre maximum de partitions distinctes pouvant exister à un instant donné en cache, et ℓ le nombre de lignes du cache. Le maintien des tailles maximales de partitions nécessite quant à lui $\lceil log_2(a+1) \rceil * P$ bits supplémentaires, avec a l'associativité du cache considéré.

Il est à noter qu'il ne peut y avoir plus de partitions différentes que de voies disponibles dans le cache, $P \leq a$. De plus, dans le pire des cas, P est égal au nombre de tâches critiques différentes dans le système ; il faut distinguer autant de partitions que de tâches critiques. Toutefois, des tâches différentes peuvent partager un même identifiant si elles ne s'exécutent pas concurremment. Par exemple, si deux tâches critiques s'exécutent sur le même cœur, elles peuvent utiliser la même partition. Un tel scénario implique toutefois que, si préemption il y a, une analyse du coût de cette dernière soit effectuée. Ce coût est borné par la taille de la partition.

La logique du cache, utilisée dans le cadre de la mise en œuvre de la politique d'éviction, doit aussi être modifiée pour assurer l'implémentation d'un cache PRETI. Ces modifications visent à permettre la sélection de la plus vieille ligne de cache dont l'identifiant de tâche est soit indéfini, soit invalide, ou correspond à l'identifiant d'une tâche en sur allocation. L'invalidation de l'identifiant d'une ligne de cache est lié à la terminaison de la tâche correspondante, ou sa préemption. Cette invalidation assure la rentrée en espace partagé des lignes de cache précédemment propriétés de la tâche terminée. Des modifications similaires de la politique de remplacement ont déjà été traitées, elles restent limitées et n'interviennent pas

sur le chemin critique [80, 73].

3.1.3 Analyse de caches partitionnés avec PRETI

Pour la validation des systèmes temps-réel, vis-à-vis de leurs contraintes temporelles, des garanties sur les performances des tâches sont requises, notamment concernant leur pire temps d'exécution. Pour les systèmes fonctionnant sur une architecture équipée de caches, cela passe par une phase d'analyse du contenu de ces derniers afin de capturer les défauts et les succès, et estimer ainsi leur contribution au pire temps d'exécution. Dans le cas de l'analyse d'un cache PRETI, il est garanti que si N voies sont allouées à une tâche, les N blocs les plus récents de cette dernière sont maintenus en cache, en dépit des interférences créées par des tâches concurrentes. Comme dans le cadre d'un partitionnement classique, l'espace privé d'une tâche dans un cache PRETI se comporte donc de manière similaire à un cache LRU privé, de taille équivalente à l'espace alloué à la dite tâche.

Quant au contenu, pour un cache PRETI, chaque tâche peut donc être analysée en isolation, sans tenir compte des interférences de ses concurrentes. Il suffit de considérer au pire un cache privé d'une largeur équivalente à l'allocation de la tâche considérée. Au mieux, la tâche peut avoir accès à l'intégralité de l'espace du cache. Les analyses classiques pour l'estimation de la contribution au pire temps d'exécution de caches privés peuvent donc être utilisées pour l'analyse de la contribution d'un cache PRETI, qu'elles visent à une estimation sûre du WCET*, worst case execution time* via analyse statique ou à une estimation probabiliste de la distribution des temps d'exécution.

Pour les méthodes basées sur des mesures du profil temporel d'une application, comme pour l'analyse statique, la tâche cible peut être exécutée en isolation en considérant un cache privé de taille équivalente à sa partition. Un environnement d'exécution valide peut être obtenu par l'exécution en amont, sur le cache PRETI, d'une tâche synthétique à laquelle est alloué le reste du cache, et dont l'objectif est simplement de remplir sa partition.

Les classifications des accès aux instructions et données *partagées* dans le cadre d'un cache PRETI suivent les mêmes restrictions que dans un cache classique (p. 40, § 2.2.3). Un cache PRETI se comporte à la fois en cache privé et en cache partagé, du fait respectivement des espaces privés et de l'espace partagé. Les restrictions relatives à la classification des accès aux données partagées dans ces deux contextes s'appliquent donc. Ces accès ne peuvent être classifiés en tant que succès dans le cas général. Les accès à un bloc mémoire partagé ne peuvent non plus être classifiés de

façon sûre comme des défauts. À l'image des méthodes proposées pour restreindre les occurrences de concurrence entre tâches [58], la durée de vie des blocs partagés dans les espaces privés rivaux peut être estimée par raffinements successifs afin de garantir leur insertion dans un espace privé. Ces méthodes impliquent toutefois la dépendance des analyses à l'ordonnancement du système.

De plus, le comportement d'un cache PRETI, et donc l'estimation de sa contribution au temps d'exécution des tâches, est indépendant de l'algorithme d'ordonnancement de tâches sous-jacent. Il s'agit d'une différence par rapport à l'utilisation d'un cache partagé non partitionné où l'intersection des durées de vie des tâches concurrentes permet de restreindre les interférences estimées [58]. Toutefois, lors de la validation d'un système, à tout instant, la somme de l'espace alloué aux tâches concurrentes sur un cache PRETI doit rester inférieure ou égale à la taille du dit cache. Une telle propriété est validée de façon implicite pour des algorithmes calculant conjointement ordonnancement et partitionnement [10, 75].

L'isolation fournie par un cache PRETI ne concerne que le contenu du cache, et plus précisément le contenu des espaces privés de chaque tâche. La possibilité d'analyser ou de mesurer le comportement temporel d'une tâche en isolation de ses concurrentes peut être remise en cause par d'autres mécanismes architecturaux comme l'arbitre de bus.

3.2 Expérimentations

PRETI vise à faciliter l'ordonnancement de systèmes critiques par une précision d'analyse accrue par rapport à des caches partagés soumis aux conflits inter-tâches. De plus, la composition à la volée d'un espace partagé accommode les tâches non critiques sans partition. Après une présentation des conditions d'expérimentations (§ 3.2.1), nous évaluons par la suite la capacité de PRETI à satisfaire à ces deux objectifs, la prédictibilité pour les tâches critiques (§ 3.2.2) et les performances pour les autres (§ 3.2.3).

3.2.1 Conditions expérimentales

Architecture considérée.

L'architecture considérée est un processeur multi-cœurs comprenant 3 cœurs exécutant chacun une tâche en parallèle. La hiérarchie mémoire est composée de deux niveaux. Les caches de premier niveau sont privés à chaque cœur et com-

prennent un cache de données supposé parfait et un cache d'instruction à correspondance directe. Le cache d'instruction de second niveau est partagé entre tous les cœurs. La taille et l'associativité des caches considérés varient selon différentes configurations détaillées dans le tableau 3.1. La taille des lignes de cache dans la hiérarchie est fixée à 32 octets. Les latences d'accès au pire cas pour les premiers niveaux, le second niveau, et la mémoire sont respectivement de 1, 7 et 21 cycles. Ces latences incluent l'accès au bus mémoire et l'arbitrage d'accès au cache partagé.

TABLE 3.1 – Configurations de caches d'instructions considérés dans l'évaluation de la politique de partitionnement PRETI.

	Taille du cache		**Associativité**	
	L1	L2	L1	L2
tiny	256B	2KB	1	8
small	512B	4KB	1	8
medium	1KB	8KB	1	8

Le cache de second niveau peut implémenter différentes politiques de partitionnement selon le scénario étudié. Les différentes configurations disponibles sont un LRU classique, un partitionnement de type PRETI, ou un partitionnement strict du cache. L'utilisation d'une hiérarchie mémoire où seuls les caches d'instructions sont un facteur de variation du pire temps d'exécution permet de réduire l'indéterminisme des estimations obtenues. Ces estimations sont donc plus proches des comportements obtenus par simulation dans les expérimentations suivantes.

Applications considérées.

Le jeu de tâches temps-réel considéré par la suite est issu de l'application debie [42], développée par la société *Space Systems Finland Ltd* (SSF). Déployée sur un satellite scientifique, elle a pour but la mesure d'impacts de micro météorites et de petits débris.

Les tâches non critiques, utilisées afin d'obtenir un système de criticité hybride, sont un sous-ensemble des *WCET benchmarks*. Ces tâches sont maintenues par le groupe de recherche de l'université de Mälardalen [34]. Elles correspondent à des schémas d'applications embarquées classiques.

Les caractéristiques des différentes applications sélectionnées sont détaillées, par jeu de tâches, dans le tableau 3.2. Pour chacune sont présentées les tailles de ses différents segments de code et de données (colonnes 2, 3, 4 et 5 respectivement

pour le code, les données initialisées, ou non initialisées, et la pile). Dans le cas des tâches temps-réel *debie*, la fréquence d'exécution (colonne 6) est aussi présentée. Elle correspond au nombre d'occurrences de la tâche par seconde.

Les fréquences d'exécution des tâches du système *debie* présentées dans le tableau 3.2 ont été augmentées par rapport à la description du système [42]. Dans le cas contraire, l'utilisation de la tâche *monitoring_task*, le rapport entre son pire temps d'exécution et sa période, est le seul facteur limitant la possibilité d'ordonnancer le système. En configuration standard si *monitoring_task* peut être ordonnancée, toutes les autres tâches tiennent sur un cœur adjacent. Dans la configuration de fréquences choisie, la pression des autres tâches est plus importante.

TABLE 3.2 – Caractéristiques, tailles des segments de code, de données (répartis en données initialisées ou non, et pile) et fréquence d'exécution, des tâches étudiées dans le cadre de l'évaluation de la politique de partitionnement PRETI.

DEBIE

Task	**Code** (octets)	**Données initialisées** (octets)	**Données non initialisées** (octets)	**Pile** (octets)	**Fréquence** (hertz)
acquisition_task	6168	5168	101904	161	400Hz
hit_trigger_handler	2684	543	218	68	1600Hz
monitoring_task	12732	564	65772	250	1Hz
tc_execution_task	13264	50	101444	164	400Hz
tc_interrupt_handler	2980	31	35648	25	4000Hz
tm_interrupt_handler	972	23	168	20	4000Hz

MALARDALEN

Task	**Code** (octets)	**Données initialisées** (octets)	**Données non initialisées** (octets)	**Pile** (octets)	**Fréquence**
bs	336	124	0	24	-
fft	3516	0	128	248	-
jfdctint	3000	0	256	84	-
minver	4540	88	432	124	-

Environnement d'analyse.

Les analyses présentées dans la suite ont été conduites à l'aide de l'analyseur Heptane. Ce dernier repose sur une méthode d'énumération implicite des

chemins (IPET) pour estimer le pire temps d'exécution (WCET) des applications analysées. La contribution temporelle de chaque référence mémoire, par rapport à la hiérarchie mémoire, est calculée en utilisant les méthodes introduites dans [40].

Seule la contribution des caches d'instructions au pire temps d'exécution est prise en compte par la suite. Les effets d'autres mécanismes architecturaux ne sont pas considérés. Cela exclut donc les anomalies temporelles liées aux interactions entre pipelines et caches ; une référence non classifiée (NC) peut être considérée de façon sûre comme toujours en défaut (AM).

Le compilateur gcc en version 4.5.2 a produit les exécutables Mips R2000/R3000 utilisés dans le cadre des expérimentations. Aucune optimisation n'a été activée lors de la compilation. Les latences d'accès aux différents niveaux de la hiérarchie sont bornées et connues. Dans le cas de l'utilisation d'un bus partagé, cela suppose par exemple l'utilisation d'une politique d'arbitrage appropriée.

Environnement de simulation.

Dans le cadre de la mesure des performances effectives des tâches étudiées, un environnement de simulation dirigé par des traces a été mis en place. Un modèle temporel simple, correspondant à celui utilisé dans le cadre des analyses, est simulé. Une unique instruction est récupérée et exécutée par cycle. L'exécution d'une tâche est bloquée dans l'évènement d'un défaut de cache, jusqu'à résolution de ce dernier. Le simulateur se concentre donc principalement sur l'estimation du comportement de la hiérarchie mémoire afin d'étudier la contribution des caches et d'obtenir un environnement similaire à celui analysé.

Pour chaque application, une unique trace d'adresses est utilisée durant les simulations. Cette dernière correspond aux références mémoire sur le pire chemin d'exécution de l'application, tel que calculé par l'analyseur Heptane.

Ordonnancement de tâches

Les tâches temps-réel étant contraintes par leurs échéances, une politique d'ordonnancement temps-réel doit être utilisée pour cette dernière. Dans le cadre de ces expérimentations, la politique NP-EDF (*non-preemptive earliest deadline first*) [47] est mise en œuvre. Chaque fois qu'une nouvelle tâche temps-réel arrive sur le système, les tâches prêtes sont ordonnées par échéance, de la plus courte à la plus lointaine. Si aucune autre tâche temps-réel n'est en cours d'exécution, celle avec la plus courte échéance est sélectionnée pour exécution, sans préemption, jusqu'à terminaison.

L'utilisation d'une telle politique non préemptive permet d'éliminer, pour les tâches critiques, l'impact des délais de préemption liés aux caches, et plus particulièrement aux caches de premier niveau dans la configuration choisie. Nous nous concentrons sur l'impact de PRETI sur les conflits inter-tâches, liés au partage du cache de second niveau sur l'architecture multi-cœur étudiée.

Dans les configurations explorées et sélectionnées par la suite, chaque tâche est allouée à un cœur unique et toutes ses instances s'exécutent sur ce même cœur. Le même cœur peut être attribué à plusieurs tâches temps-réel différentes. L'ordonnancement NP-EDF s'applique et est validé indépendamment sur chaque cœur. Cette configuration d'ordonnancement correspond à celles produites par des algorithmes comme IA3 [75] et PDPA [10] qui effectuent conjointement le partitionnement des tâches et du cache entre cœurs.

Les tâches non critiques, issues du jeu de tâches Mälardalen, s'exécutent en tâche de fond, quand aucune tâche temps-réel n'est prête à s'exécuter. Elles sont préemptées dès l'arrivée d'une tâche temps-réel pour laisser la main et, de fait, sont soumises aux délais de préemptions.

3.2.2 Impact de PRETI sur l'ordonnançabilité des systèmes

Une première série d'expérimentations se focalise sur l'impact de PRETI sur le temps d'exécution au pire cas statiquement calculé de tâches critiques. Nous étudions la capacité de la politique de remplacement PRETI à faciliter l'ordonnancement de systèmes temps-réel par rapport à un cache partagé classique (SHARED).

À cette fin, nous explorons exhaustivement les scénarios valides d'allocation de cache aux différentes tâches et de tâches aux différents cœurs. Cette exploration nous affranchit de la politique de partitionnement sous-jacente. Pour chaque configuration de cache et de politique de remplacement, la meilleure solution est sélectionnée, c'est-à-dire celle garantissant l'ordonnançabilité du système à fréquence de processeur minimale. Il est supposé, pour des questions de simplicité et afin de conserver la même latence d'accès à la mémoire, que la fréquence de la mémoire principale varie avec celle du processeur.

Pour chacune des configurations, taille de cache (de haut en bas) et politique de remplacement (SHARED pour le LRU classique, en colonne 2 et PRETI en colonne 3), le tableau 3.3 présente la fréquence minimale d'ordonnancement du jeu de tâches debie. Le tableau 3.4 présente les allocations correspondantes, allocation de tâches aux cœurs d'exécution (colonne 2 et colonne 3 respectivement pour SHARED et PRETI) et allocation de cache aux tâches (colonne 4), pour atteindre

l'ordonnançabilité dans ces différentes configurations (de haut en bas).

TABLE 3.3 – Fréquence minimale d'ordonnancement du système debie sous différentes configurations de caches d'instructions et sous la politique de remplacement LRU classique (SHARED) ou PRETI.

	Politique de remplacement	
	SHARED	PRETI
tiny	723Mhz	669Mhz
small	670Mhz	613Mhz
medium	640Mhz	426Mhz

Dans toutes les configurations de cache explorées, la politique de remplacement PRETI permet d'ordonnancer le système considéré à une fréquence plus petite qu'une politique LRU classique, sujette aux conflits inter-tâches. En effet, dans la configuration SHARED, ces conflits sont tels que le cache partagé ne peut être considéré de façon sûre comme utilisable par les tâches analysées. En utilisant la politique de remplacement PRETI, une portion du cache peut être dédiée aux tâches les plus critiques et ainsi réduire leur utilisation du processeur, le rapport entre leur période et leur pire temps d'exécution.

À mesure que croît la taille du cache partagé, de la configuration *tiny* à la configuration *medium*, plus de ressources sont disponibles pour le système. Dans le cas de SHARED, les conflits sont tels que le bénéfice de cet incrément pour les tâches du système debie ne peut être capturé de façon sûre. Au contraire un cache PRETI plus grand permet des partitions plus grandes et donc accommodant plus facilement les tâches. L'écart entre la politique PRETI et la politique SHARED se creuse donc avec les grandes configurations de cache, aux dépens de la politique SHARED.

L'allocation des tâches aux différents cœurs tend à rester la même dans les différentes configurations (voir tableau 3.4). Dans un système non préemptif, les tâches se regroupent notamment en fonction de leur fréquence. La tâche *monitoring_task* se retrouve seule. Pour un cache partitionné, la tâche *acquisition_task* se retrouve aussi en possession exclusive de son cœur. En effet, le bénéfice de l'allocation d'une portion du cache à la tâche *hit_trigger_handler* permet de la placer avec les tâches de plus haute fréquence, et avec *acquisition_task* dans le cas contraire (configuration SHARED).

Concernant l'allocation du cache aux différentes tâches, dans le cas de la politique PRETI, les configurations permettant d'atteindre la fréquence minimale d'ordonnancement varient avec la configuration de cache. Notamment, la tâche *mo-*

nitoring_task obtient des bénéfices plus importants d'un espace dédié avec les configurations *small* et *medium*. En configuration *tiny*, les bénéfices pour cette tâche du cache partagé sont maigres et c'est la tâche *hit_trigger_handler* qui bénéficie donc de la partition la plus importante. Les tâches *tc_execution_task*, *tc_interrupt_task*, *tm_interrupt_task*, montrent peu de bénéfice à l'utilisation d'un cache de second niveau et s'imposent donc peu dans le choix du partitionnement du cache.

Les conflits inter-tâches nuisent à la précision des analyses de contenu de caches partagés et donc à la précision des estimations de pire temps d'exécution en contexte multi-cœur. Il en résulte une sur-allocation de ressources, symbolisée ici par la fréquence du processeur cible, afin de permettre l'ordonnancement de systèmes concurrents. L'utilisation du partitionnement, par exemple avec la politique de remplacement PRETI, permet de réduire ces conflits au bénéfice de l'ordonnançabilité des systèmes.

3.2.3 Impact de PRETI sur les performances mesurées des tâches

Dans cette série d'expérimentations, nous observons l'impact de PRETI sur le temps d'exécution mesuré de tâches non critiques en système de criticité mixtes. En particulier, nous montrons la capacité de PRETI à offrir des performances satisfaisantes pour les tâches non critiques, une fois garantie l'ordonnançabilité d'un jeu de tâches critiques.

Dans cet objectif, nous simulons le système debie en utilisant les partitionnements et fréquences d'exécution du processeur déterminées précédemment en tableaux 3.4 et 3.3 respectivement. Ces allocations garantissent l'ordonnançabilité du système. Afin de composer un système de criticité mixte, différents jeux de tâches non critiques sont considérés en concurrence avec debie dans chaque configuration. Dans tous les cas et quel que soit le partitionnement du cache L2, la fréquence d'exécution du processeur ou le jeu de tâches concurrentes, le système est simulé pour un milliard de cycles.

Quatre scénarios pour la gestion du cache partagé L2 sont ici comparés :

– SHARED correspond à l'utilisation d'un cache partagé classique, non partitionné. Le cache L2 est géré selon une politique LRU. Toutefois, une telle configuration ne garantit pas la sûreté du système debie aux fréquences simulées.

TABLE 3.4 – Allocation aux différentes tâches des cœurs d'exécution et de l'espace du cache autorisant l'ordonnancement du jeu de tâches debie à fréquence minimale, présentée dans le tableau 3.3, en utilisant une politique de remplacement LRU classique (SHARED) ou une politique PRETI.

tiny

Application	**Allocation**		
	SHARED	PRETI	
	cœur	cœur	Cache
acquisition_task	2	2	0
hit_trigger_handler	0	0	5
monitoring_task	1	1	3
tc_execution_task	0	0	0
tc_interrupt_task	0	0	0
tm_interrupt_task	0	0	0

small

Application	**Allocation**		
	SHARED	PRETI	
	cœur	cœur	Cache
acquisition_task	2	2	0
hit_trigger_handler	2	0	3
monitoring_task	1	1	5
tc_execution_task	0	0	0
tc_interrupt_task	0	0	0
tm_interrupt_task	0	0	0

medium

Application	**Allocation**		
	SHARED	PRETI	
	cœur	cœur	Cache
acquisition_task	2	2	0
hit_trigger_handler	2	0	3
monitoring_task	1	1	3
tc_execution_task	0	0	0
tc_interrupt_task	0	0	0
tm_interrupt_task	0	0	0

- STRICT-CORE correspond à l'utilisation d'un cache strictement partitionné. Les partitions sont allouées à la granularité du cœur d'exécution. Toutes les tâches s'exécutant sur un cœur, critiques et non critiques, utilisent sa partition.
- PRETI-CORE correspond à l'utilisation d'un cache PRETI. Comme pour la configuration STRICT-CORE, chaque partition est allouée à un cœur d'exécution. Les tâches sur un cœur sans partition utilisent l'espace partagé.
- PRETI-THREAD correspond aussi à l'utilisation d'un cache PRETI. Les partitions sont allouées à chaque tâche. Les tâches sans partition utilisent donc l'espace partagé.

La taille de la partition allouée à chaque cœur ou tâche pour chacun de ces scénarios est présentée dans le tableau 3.5. Ainsi que précédemment mentionné, ces allocations correspondent à celles dérivées dans la section précédente et garantissent l'ordonnançabilité du système debie. Dans le cas de la configuration STRICT-CORE, les voies non requises pour cette garantie sont allouées au cœur restant.

L'IPC (*instructions per cycle*) des tâches non critiques bs, jfdctint et minver (respectivement jfcdtint, ludcmp et fft) sous les différents scénarios étudiés, et normalisé sur leur IPC sous la configuration SHARED avec une hiérarchie de taille équivalente, est présenté en Figure 3.1 (respectivement Figure 3.2). De haut en bas, Figures 3.1a, 3.1b et 3.1c, sont présentés les résultats pour les différentes tailles de hiérarchie mémoire considérées. Dans chaque cas, les trois tâches non critiques sont concurrentes entre elles et concurrentes aux tâches de *debie*. D'autres jeux de tâches non critiques ont été explorés mais les comportements observés sont similaires à ceux présentés dans ce document.

La comparaison des configurations STRICT-CORE et PRETI-CORE permet d'observer le comportement d'un cache strictement partitionné par rapport à un cache PRETI sous un partitionnement équivalent. Dans toutes les configurations explorées, le cache PRETI se comporte, pour les tâches non critiques, aussi bien (Figure 3.1a pour jfdctint) voire mieux (fft en Figure 3.2b) que le cache strictement partitionné. Les principales différences tiennent à l'espace partagé et, dans la configuration *medium* à l'allocation, ou non, d'une partition au cœur 2.

Pour les petites tailles de cache, *small* et *tiny*, le partitionnement strict du cache a un fort impact négatif sur les performances de la tâche allouée au second cœur (fft en Figure 3.2a). En effet, l'intégralité du cache est répartie entre les cœurs 0 et 1 ; les tâches sur le cœur 2 ne peuvent utiliser le cache dans le scénario STRICT-CORE. Dans les scénarios basés sur un cache PRETI, l'espace partagé, disponible

TABLE 3.5 – Scénarios d'allocation du cache partagé à chaque tâche critique ou cœur pour chaque configuration de simulation, taille de la hiérarchie mémoire et politique de remplacement du cache partagé.

tiny

Configuration de cache	**cœur/Tâche**	**Partition**
STRICT-CORE	cœur 0	5 voies
	cœur 1	3 voies
	cœur 2	0 voie
PRETI-CORE	cœur 0	5 voies
	cœur 1	3 voies
	cœur 2	0 voie
PRETI-THREAD	hit_trigger_handler	5 voies
	monitoring_task	3 voies

small

Configuration de cache	**cœur/Tâche**	**Partition**
STRICT-CORE	cœur 0	3 voies
	cœur 1	5 voies
	cœur 2	0 voie
PRETI-CORE	cœur 0	3 voies
	cœur 1	5 voies
	cœur 2	0 voie
PRETI-THREAD	hit_trigger_handler	3 voies
	monitoring_task	5 voies

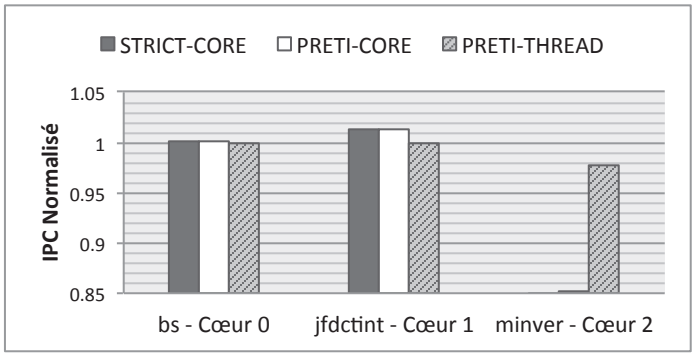
medium

Configuration de cache	**cœur/Tâche**	**Partition**
STRICT-CORE	cœur 0	3 voies
	cœur 1	3 voies
	cœur 2	2 voie
PRETI-CORE	cœur 0	3 voies
	cœur 1	3 voies
	cœur 2	0 voie
PRETI-THREAD	hit_trigger_handler	3 voies
	monitoring_task	3 voies

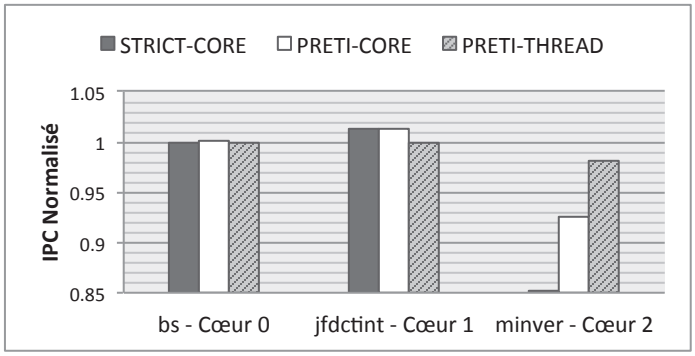
par exemple suite à la terminaison d'une tâche, permet à ces tâches de bénéficier du cache, même pour de courts instants. Pour ce cœur, le bénéfice d'une allocation moins agressive du cache, comme PRETI-THREAD par opposition à PRETI-CORE, est encore plus marqué (voir Figure 3.2a).

Toutefois, un partitionnement strict minimal du cache, pour simplement garantir l'ordonnançabilité du système critique étudié, peut avoir un impact négatif pour les performances des tâches non critiques. Par exemple, ludcmp, en Figure 3.2, bénéficie fortement de la partition allouée au cœur 1 dans les scénarios STRICT-CORE et PRETI-CORE. L'isolation fournie par cette allocation protège en effet la tâche des conflits inter-tâches qui peuvent impacter ses performances. La tâche se comporte mieux quand elle dispose d'une partition, même réduite (Figure 3.2b), que lorsqu'elle doit partager le cache avec l'intégralité des tâches du système.

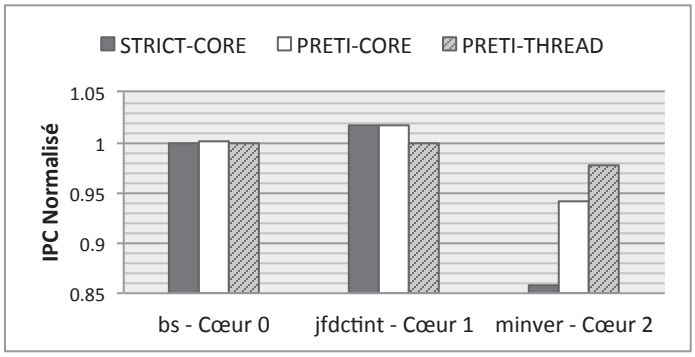
Comparée à une implémentation stricte classique, à partitionnement identique, la politique de remplacement PRETI offre des performances similaires, souvent meilleures. Toutefois, un cache PRETI offre l'avantage de la polyvalence. À partir des mêmes contraintes garantissant la validation d'un système, différentes stratégies d'allocations peuvent être explorées, plus ou moins agressives, par exemple afin de mitiger les phénomènes de famine ou de réduire les conflits inter-tâches en permettant une isolation partielle.



(a) Performances des tâches non critiques sous la configuration de cache *tiny*

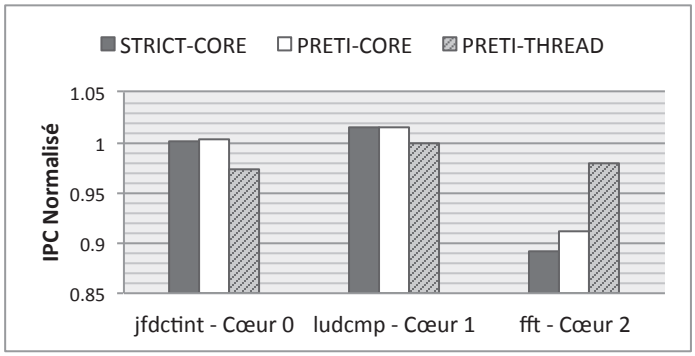


(b) Performances des tâches non critiques sous la configuration de cache *small*

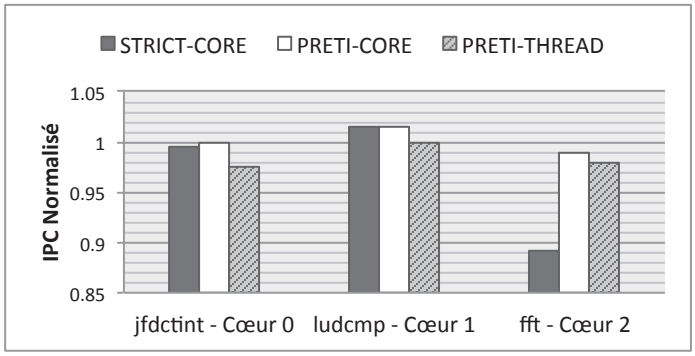


(c) Performances des tâches non critiques sous la configuration de cache *medium*

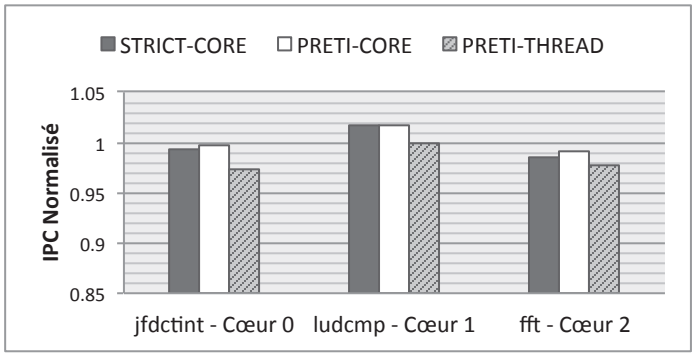
FIGURE 3.1 – IPC normalisé des tâches non critiques, bs, jfdctint et minver, allouées respectivement aux cœurs 0, 1 et 2, en concurrence avec le jeu de tâche debie, sous différentes tailles de cache (tableau 3.1) et politiques de remplacement. L'IPC avec un cache partagé classique sous une configuration de tâches et de taille similaire est utilisé comme base



(a) Performances des tâches non critiques sous la configuration de cache *tiny*



(b) Performances des tâches non critiques sous la configuration de cache *small*



(c) Performances des tâches non critiques sous la configuration de cache *medium*

FIGURE 3.2 – IPC normalisé des tâches non critiques, jfdctint, ludcmp et fft, allouées respectivement aux cœurs 0, 1 et 2, en concurrence avec le jeu de tâche debie, sous différentes tailles de cache (tableau 3.1) et politiques de remplacement. L'IPC avec un cache partagé classique sous une configuration de tâches et de taille similaire est utilisé comme base.

3.3 Étendre le comportement des caches PRETI

Nous avons précédemment présenté et évalué PRETI dans son expression la plus simple. Nous proposons maintenant différentes solutions matérielles pour améliorer la portée et le contexte du mécanisme.

Le mécanisme permet l'analyse de l'espace privé de tâches en isolation. Par le biais de données partagées une tâche peut impacter le contenu de l'espace privé d'une rivale, comme un cache privé sous l'impact d'une politique de cohérence.

L'espace partagé, composé des lignes non allouées ou non utilisées par leur propriétaire, accommode les tâches les moins critiques. Plus le volume d'espace partagé reste important, plus les tâches peuvent en bénéficier. Il peut donc être intéressant de retarder l'entrée de lignes dans l'espace privé au plus tard.

Le mécanisme a été présenté sur la base de la politique de remplacement LRU. La notion d'âge utilisée pour identifier les blocs les plus conservés dans un espace privé peut s'étendre à d'autres politiques de remplacement. En effet, la politique LRU peut s'avérer coûteuse à implémenter pour de grandes associativités.

3.3.1 Restriction des accès aux données partagées

Les données partagées, ainsi que mentionnées précédemment, posent des problèmes de prédictibilité pour les analyses de cache. Dans le cadre des caches privés, les conflits intra-tâches ne sont plus le seul facteur d'éviction ; les protocoles de cohérence doivent être considérés. Ces données partagées souffrent aussi des conflits inter-tâches au niveau des caches partagés. Les analyses doivent aussi considérer les effets constructifs du partage de données. Une tâche peut charger un bloc au bénéfice des autres. Nous nous intéressons dans la suite à la gestion des accès aux données partagées dans le cadre de caches PRETI.

Dans la politique de remplacement PRETI, telle que présentée précédemment, lorsqu'une tâche accède à un bloc mémoire, ce dernier est inséré sans condition en tête de son espace privé. Si ce bloc est partagé entre différentes tâches, il peut donc être réquisitionné depuis l'espace privé d'une autre tâche. En plus de briser la propriété d'isolation entre des espaces privés distincts, ce comportement pose le même problème de l'éviction prématurée des données partagés du fait de tâches concurrentes. De plus, l'espace privé où a eu lieu la réquisition se retrouve réduit.

Des comportements spécifiques sont requis pour assurer l'isolation d'espaces privés concurrents. Nous présentons par la suite deux méthodes permettant d'assurer cette propriété sur la base de l'identification dynamique des données partagées

ou leur connaissance à priori.

Court-circuitage des accès partagés

Pour assurer l'isolation d'espaces privés concurrents, le comportement de la politique de remplacement PRETI doit être modifié. En particulier, lors d'un succès, si le bloc accédé se trouve dans l'espace privé d'une autre tâche, l'état du cache reste inchangé. Aucune promotion ou mouvement logique dans le cache n'a lieu si le bloc accédé se trouve dans l'espace privé d'une tâche concurrente. Dans les autres scénarios, la politique de promotion reste inchangée et un bloc accédé en espace partagé ou dans l'espace privé de la tâche en émettant la requête est marqué comme étant le plus récemment utilisé.

La mise en œuvre de ce comportement requiert une modification de la logique d'accès au cache afin de tester l'appartenance du bloc accédé à un espace privé rival. Ces modifications ne sont pas sur le chemin critique et concernent la mise à jour des structures de contrôle du cache. La donnée accédée peut être servie immédiatement si elle est présente, laissant ainsi la latence en cas de succès inchangée. De plus, la connaissance des données partagées à priori n'est pas requise. Leur identification se fait uniquement quand leur prise en compte requiert un comportement particulier, c'est-à-dire quand elles se trouvent dans un espace privé concurrent.

Restriction des espaces d'insertion autorisés

Une instruction, si elle est connue pour accéder à des données partagées, pourrait être configurée afin de ne pas modifier l'espace privé de sa tâche à l'exécution. Il s'agit d'une restriction plus forte que la solution précédente. Du point de vue de l'analyse, ces instructions n'ont aucun effet sur le contenu de cache estimé. Toutefois, cette mise en oeuvre requiert la connaissance des données partagées par le système ou une modification du jeu d'instructions pour marquer les accès à des données partagées.

Dans cette configuration, les données partagées sont interdites dans les espaces privés dédiés à une tâche. Elles peuvent toutefois être insérées et promues au sein de l'espace partagé comme les blocs manipulés par des tâches non critiques. Les données partagées les plus critiques ou utilisées peuvent aussi se voir allouer un espace en cache. Cet espace est alors sous la responsabilité partagée, notamment quant à sa libération, des différentes tâches utilisant les données partagées qui y sont stockées.

3.3.2 Contrôle de la croissance des partitions

Dans un cache PRETI, l'espace partagé peut être utilisé par toutes les tâches du système, notamment les tâches non critiques, afin de maximiser le volume de cache utile. Cet espace est réduit à mesure que les tâches réquisitionnent des lignes au profit de leurs espaces privés. Les lignes ainsi allouées à un espace privé ne sont libérées qu'à la terminaison ou, selon la configuration, la préemption de son propriétaire. Les bénéfices qu'une tâche tire de son espace privé peuvent toutefois s'avérer moindre que ceux que d'autres tireraient de l'espace partagé.

Le mécanisme de bypass permet de prévenir toute modification logique du cache par une instruction. Sur cette base, une référence mémoire qui court-circuite le cache n'a donc aucun impact sur son espace privé ou sur l'espace partagé. Il serait donc intéressant de prévenir la croissance d'un espace privé sous l'impulsion de certaines références mémoire si l'on n'est certain que la contribution de ces dernières n'apporte aucun bénéfice. La mise en œuvre et les bénéfices d'une approche similaire ont déjà été démontrés au chapitre précédent, afin de réduire la pression des tâches sur les caches partagés. Toutefois, prévenir l'insertion d'un bloc en cache par une tâche, à défaut d'impacter son pire temps d'exécution estimé peut avoir un effet destructif sur son temps moyen d'exécution.

Une utilisation du mécanisme du bypass tel qu'il a été précédemment défini est donc une solution trop radicale pour pallier au problème de la croissance des partitions de tâches critiques. En effet le bypass interdit les modifications au niveau du cache. Un mécanisme plus approprié préviendrait l'insertion ou la promotion de blocs au sein de l'espace privé de la tâche concernée. Un accès bypassant le cache insérerait le bloc cible dans l'espace partagé, comme si il était émis par une tâche sans partition allouée. Du point de vue de l'analyse, ce mécanisme se comporte comme un mécanisme de bypass classique. À l'exécution toutefois, le bloc est inséré en cache, avec une durée de vie plus courte, et peut donc être réutilisé.

3.3.3 Extensions à d'autres politiques de remplacement, au delà du LRU

La politique de remplacement PRETI assure aux tâches critiques l'allocation d'un volume de cache connu géré selon la politique LRU, favorisée pour sa prédictibilité [81] et la prévalence des travaux existants pour sa prise en compte [94]. Cette partition, indépendante des effets de tâches concurrentes, peut être analysée en utilisant des méthodes classiques pour caches privés dont la taille correspond à

celle de la portion de cache allouée à la tâche considérée.

Le potentiel, à des fins d'analyse de contenu de caches, d'autres politiques de remplacement a été étudié, notamment pour les politiques *first-in first-out* [31], *not recently used* [32], et *pseudo*-LRU [30]. Si, à configuration de cache équivalente, les estimations obtenues ne sont pas aussi précises qu'en utilisant une politique LRU, ces politiques offrent l'avantage d'être moins coûteuses à mettre en œuvre, en particulier pour de grandes associativités.

Pour assurer un comportement compatible avec un cache PRETI, différentes propriétés doivent être maintenues. Le partitionnement doit fournir la protection contre les conflits inter-tâches ; l'état d'une partition ne doit pas être soumis aux aléas de tâches concurrentes. Un changement de l'état de cache initié par une tâche ne peut impacter l'ordre relatif des blocs d'espaces privés concurrents. Cet ordre est défini par la politique de remplacement sous-jacente, par exemple la politique FIFO. L'objectif est d'assurer qu'une partition de taille donnée se comporte de manière identique à un cache privé de taille équivalente. La validation de ces contraintes dans le cadre des politiques FIFO, NRU et PLRU est discutée en annexe A.

Conclusion

Dans le présent chapitre, nous avons introduit la politique de partitionnement PRETI. Celle-ci, sur la base de simples modifications de politiques de remplacement existantes, offre des avantages majeurs dans le cadre de systèmes temps-réel critiques concurrents notamment sur architectures multi-cœurs. Par rapport à un cache partagé classique, un cache PRETI permet le calcul de bornes plus précises du pire temps d'exécution de tâches critiques. Cette précision facilite l'ordonnancement de systèmes critiques.

PRETI se démarque aussi de mécanismes de partitionnement plus stricts. La composition dynamique par la politique PRETI d'un espace partagé offre à toutes les tâches utilisant le cache de bonnes performances, proches de celles obtenues en utilisant un cache LRU partagé. Grâce à cette mutualisation de l'espace non alloué, différentes allocations valides peuvent être dérivées sur la base des mêmes contraintes garantissant l'ordonnancement d'un système. Cette polyvalence permet de mitiger les pertes de performances liées aux conflits inter-tâches ou à la non allocation de ressources.

Perspectives

Évaluation en système temps-réel souples. Le mécanisme de partitionnement PRETI a été présenté et évalué dans le cadre des systèmes temps-réel strict, pour lesquels le manquement à une échéance peut avoir des conséquences importantes. Dans le cadre des systèmes temps-réel souples, un dépassement d'échéance implique des pertes plus limitées, par exemple une réduction de la qualité de service.

La politique de remplacement PRETI peut bénéficier à ces systèmes. Par l'allocation d'espace privés aux tâches du système, ces dernières peuvent être partiellement isolées des effets indésirables de leurs concurrentes sur le cache. Cette isolation permet ainsi une réduction de la variabilité de leur temps d'exécution. Toutefois, l'intérêt de ces méthodes dans un tel contexte reste encore à évaluer.

Les méthodes temps-réel souples se prêtent à la validation par mesure du temps d'exécution des applications. L'intégration d'un cache PRETI dans un environnement de mesure est un autre aspect non étudié ici. Une approche simple pour des mesures serait de préchauffer le cache PRETI avec une tâche synthétique occupant l'espace du cache non alloué à la tâche mesurée.

Politiques de gestion de l'espace partagé. Dans le cache PRETI tel que défini précédemment, l'espace partagé est géré selon la politique de remplacement LRU sous-jacente. L'espace partagé est soumis aux mêmes limitations que les caches partagés classiques. En particulier, certaines applications de type *streaming* manipulent de larges volumes de données à faible localité temporelle et spatiale. Elles évincent donc du cache des données utiles sans les y remplacer par d'autres. Dans le cadre d'architectures généralistes, de nombreux mécanismes, basés sur des modifications de la politique de remplacement, ont été proposés pour minimiser l'impact de ce phénomène.

L'espace partagé construit par un cache PRETI pourrait bénéficier de telles politiques d'équité afin d'y réduire l'impact des conflits inter-tâches. La politique de gestion choisie doit être d'un coût d'implémentation faible. De plus, elle ne peut intervenir dans la gestion des espaces privés afin de pouvoir assurer statiquement leur disponibilité aux tâches critiques.

Une autre méthode pour réduire la pression sur l'espace partagé est de prévenir l'utilisation de ce dernier par certaines tâches, en les restreignant par exemple à leur seule partition comme dans le cadre d'un partitionnement strict. La décision de restreindre une tâche à sa seule partition et de lui interdire l'accès à l'espace partagé peut être prise dynamiquement par exemple en estimant les bénéfices attendus de ce comportement [46]. Statiquement, il est toujours garanti à chaque tâche qu'elle aura au minimum accès à l'espace alloué.

À l'inverse, un mécanisme intelligent pourrait augmenter les partitions allouées à certaines tâches si le bénéfice global pour le système dépasse la perte engendrée par un espace partagé réduit. Ces allocations seraient un supplément dynamique à celles requises pour la validation du système et en aucun cas ne se supplanteraient à ces dernières.

Algorithme d'allocation de cache PRETI. Les algorithmes de partitionnement [84, 68] visent à diviser le cache entre différentes tâches afin d'assurer la validité du système vis-à-vis de ses contraintes temporelles. Certains dédiés aux systèmes multi-cœurs effectuent en simultané la répartition des tâches entre les différents cœurs [75, 10]. Toutefois, ces algorithmes tendent à produire des partitionnements complets de l'espace du cache afin de minimiser par exemple la charge globale du système.

Dans le cadre d'un cache PRETI, l'espace non alloué peut bénéficier à toutes les tâches du système. Il est donc intéressant d'allouer simplement les ressources nécessaires pour assurer la validation et éventuellement la robustesse de l'ordon-

nancement du système. Les ressources non allouées sont dans tous les cas mutualisées au sein de l'espace partagé au bénéfice de toutes les tâches faisant usage du cache. Le point de vue adopté est donc différent.

La principale difficulté d'un tel algorithme est toutefois de définir une métrique à optimiser en conjonction avec l'obtention d'un système valide. Comme observé expérimentalement, pour un cache PRETI, les bénéfices d'une même configuration varient en fonction des tâches non critiques impliquées et de la pression qu'elles marquent sur l'espace partagé. Les tâches critiques ne doivent donc pas être les seules impliquées lorsqu'est décidée l'allocation du cache.

Conclusion

L'étude et l'estimation du comportement temporel au pire cas de tâches temps-réel plongées en environnement multi-cœur constitue le cœur de ce document. Le principal défi pour une telle architecture, par rapport à une architecture uni-cœur, réside en la prise en compte des conflits inter-tâches liés à la concurrence d'utilisation des ressources partagées entre cœurs. En particulier, nous nous sommes intéressés aux conflits au niveau des caches partagés de la hiérarchie mémoire. Les méthodes existantes se focalisent sur les caches d'instructions ; les méthodes permettant l'intégration de niveaux de caches de données au sein d'une hiérarchie, même en contexte uni-cœur, manquent.

Afin de permettre la prise en compte de hiérarchies mémoire plus riches, nous nous sommes dans un premier temps concentrés sur l'extension des cadres d'analyse existants [39, 40] avec des méthodes permettant la prise en compte de caches de données. Nos expérimentations valident l'intérêt, pour la précision des estimations temporelles, de la prise en compte de la hiérarchie mémoire complète. Toutefois, une tâche analysée l'est en isolation des effets de concurrentes notamment exécutées sur d'autres cœurs.

Sur la base de l'abstraction de la connaissance des occurrences d'accès de tâches rivales, abstraction précédemment évaluée dans le cadre des caches d'instructions [37, 58], nous avons présenté l'estimation des interférences subies au niveau des caches de données partagés. Les résultats expérimentaux montrent que ces abstractions, nécessaires pour des raisons de complexité, se font au détriment de la précision des estimations temporelles obtenues. Pour des raisons de sûreté, l'intégralité du cache partagé peut être supposée occupée par des tâches rivales à celle analysée.

Pour réduire la pression de chaque tâche sur les niveaux de caches partagés et les interférences auxquelles elle soumet ses rivales, nous avons étudié le potentiel du mécanisme de bypass qui prévient la modification du cache par des instructions sélectionnées. Nos heuristiques pour le bypass sont basées sur la capture de la

réutilisation des données chargées par chaque instruction. Si nulle réutilisation n'apparaît, le bypass permet d'empêcher l'instruction de contribuer au contenu de cache et donc aux conflits. L'intérêt de ces heuristiques s'est validé durant nos expérimentations.

La politique de remplacement PRETI constitue la seconde contribution majeure de ce document. En tant que mécanisme de partitionnement, cette dernière permet l'allocation d'un espace de cache dédié à chaque tâche critique. Le contenu de cette partition est garanti libre de conflits inter-tâches. Il peut être analysé en utilisant des méthodes pour caches privés. La précision des estimations temporelles obtenues facilite la validation des systèmes critiques reposant sur un cache PRETI, par rapport à un cache partagé classique.

La flexibilité de la politique de remplacement PRETI permet aussi la composition de systèmes de criticité hybride regroupant tâches critiques et non critiques. La composition d'un espace partagé avec les lignes non allouées ou non utilisées autorise en effet l'utilisation de l'intégralité de l'espace du cache au bénéfice des performances au temps moyen de toutes les tâches du système.

Perspectives

Vers la définition d'une architecture prédictible. Les travaux ici présentés se concentrent sur la considération d'une architecture mémoire similaire à celles présentes dans les architectures multi-cœurs généralistes. Ces architectures s'attachent à l'amélioration du temps d'exécution moyen. Les mécanismes mis en œuvre reposent sur l'évaluation dynamique du comportement des tâches et s'avèrent donc peu prédictibles, n'autorisant que des estimations imprécises du pire temps d'exécution.

Dans le cadre des systèmes temps-réel critiques, en lieu et place d'une architecture dirigée par les performances moyennes, la définition d'une architecture prédictible temporellement est favorable. Une architecture prédictible offre des facilités pour l'analyse statique du comportement des tâches qu'elle héberge, en réduisant la variabilité temporelle du mécanisme ciblé ou bien en visant à un contrôle accru de son comportement.

Néanmoins, un comportement défini en toute ou partie statiquement nécessite une plus grande implication des outils d'analyse et de la chaîne de compilation dans la sélection de ce comportement et son application. Pour exemple, l'utilisation de *scratchpads* en lieu et place de caches offre un contrôle total sur le chargement de blocs en hiérarchie mémoire permettant de garantir aisément succès et défauts. En

échange, la tâche ou le système sont responsables du chargement et du positionnement de ses blocs dans le *scratchpad* ; une phase de sélection des blocs retenus et de leur point d'insertion est requise.

Une architecture dédiée aux systèmes temps-réel pose donc le problème inverse et risque de ne pas concilier prédictibilité et performances temps moyen, critère important dans les systèmes hybrides. Il est donc important de définir des mécanismes architecturaux permettant un contrôle minimum, pour les tâches critiques, tout en fonctionnant de façon efficace et transparente pour des tâches non critiques concurrentes. La simplicité et un coût d'implémentation réduit jouent aussi un rôle prépondérant dans l'intégration de mécanismes dédiés au temps-réel dans des architectures standards.

Analyse et ordonnancement conjoint. L'estimation des conflits subis par une tâche, pour l'analyse de niveaux de caches partagés, repose sur l'abstraction de l'ordre et du nombre d'occurrences des accès des tâches rivales. Seuls sont conservés des tâches rivales les blocs accédés, principal facteur de conflits dans le cadre des caches partagés. Les analyses de contenu reposent donc sur une information minimale garantissant toutefois la sûreté des estimations temporelles en découlant.

L'ordonnancement du système considéré qui arrange les tâches entre elles dans le temps est aussi implicitement ignoré. Cette inconsidération dégage l'étape d'analyse des tâches de celle de validation du système. Les tâches peuvent être considérées à partir d'une connaissance partielle de leur environnement, nommément leurs rivales. Du fait de cette indépendance, l'environnement logiciel n'est pas contraint par les méthodes d'analyse.

Néanmoins, comme suggéré précédemment dans le cadre des données partagées, une prise en compte plus approfondie de l'environnement logiciel des tâches, voire sa restriction à des modèles fortement contraints [58], permet de préciser les estimations temporelles obtenues par analyse statique. L'implication accrue de l'environnement logiciel au même titre que du matériel est donc une piste importante pour la prédictibilité des systèmes temps-réel.

Une solution dans cette direction est la modélisation et la résolution conjointes du problème de l'ordonnancement d'un système et de l'estimation du comportement temporel de ses tâches. L'avantage de cette méthode est de pouvoir intégrer précisément la connaissance des tâches rivales dans le calcul des conflits.

Annexe A

Politiques de remplacement basées sur PRETI

Peu de propositions existent pour une mise en œuvre de partitionnement pour les politiques de remplacement FIFO, NRU et PLRU. Les travaux existants [49], dans le cadre de la politique NRU ou de la politique PLRU, s'ils permettent la conservation en cache des blocs d'une tâche, dans la limite de sa partition, ne répondent pas aux critères présentés dans le chapitre 3. L'état interne d'une partition peut être impacté par les accès de tâches concurrentes, précipitant notamment l'éviction de blocs hors de cette partition par rapport à un cache privé de taille équivalente.

FIFO-PRETI

La politique de remplacement FIFO (*first-in first-out*) ordonne les blocs de chaque ensemble selon leur ordre d'entrée dans le cache. Le bloc le plus récemment inséré est donc en tête (*last-in*), et le moins récemment inséré se trouve en fin de file (*first-in*). Lorsqu'un bloc doit être évincé, c'est ce dernier, le *first-in*, qui est sélectionné. La principale différence avec une politique de remplacement LRU est donc l'absence d'une politique de promotion qui amène un bloc en tête de file lors d'un succès.

Du fait de cette proximité avec la politique de remplacement LRU, la définition d'une politique de remplacement FIFO-PRETI est similaire au mécanisme introduit précédemment. La politique d'insertion est inchangée, un bloc est inséré en tête de la file, en position *last-in*, et entre dans l'espace privé de la tâche en ayant fait la requête. La politique d'éviction, au lieu de sélectionner le bloc en fin de file, le *first-in*, sélectionne le bloc le moins récemment inséré qui appartient à l'espace

partagé, ou appartient à l'espace privé de la tâche provoquant le défaut, si cet espace a atteint sa taille allouée.

MRU-PRETI

Une analyse de contenu de cache a été définie pour une implémentation spécifique de la politique NRU [32], telle qu'elle apparaît sur les architectures Intel Nehalem [21]. Pour chaque ligne d'un ensemble, un bit RU est positionné pour indiquer que le bloc qu'elle contient a été récemment utilisé. Lors d'une insertion ou d'une promotion, le bit de la ligne contenant le bloc ciblé est positionné. Si suite à cette accès tous les bits RU de l'ensemble cible se retrouvent positionnés, ils sont remis à zéro, exception faite de la ligne contenant le bloc cible. Cette opération de remise à zéro est nommée un *Global Flop*.

Lorsque l'éviction d'un bloc est requise, dans un ensemble de cache donné, les lignes de ce dernier sont parcourues dans un ordre connu et fixé par l'implémentation. La première ligne dont le bit RU n'est pas positionné est sélectionnée. L'ordre logique entre les blocs dépend donc en premier lieu de la position du bit RU de leur ligne ; les lignes dont le bit RU est positionné sont plus récentes que celles pour lesquelles il n'est pas positionné. À valeur équivalente du bit RU, c'est l'ordre du parcours des lignes pour la politique d'éviction qui ordonne les blocs.

La mise en œuvre d'une politique de remplacement du type PRETI requiert plusieurs modifications de ce comportement. Tout d'abord, l'opération de *flop*, autant que la décision de l'effectuer, doit être effectuée de façon locale pour chacun des espaces modifiés par un accès au cache. Dans le cas contraire, toute tâche peut modifier les bits RU d'espaces privés concurrents.

Le *task id* associé à une ligne de cache, utilisé pour identifier l'espace dans laquelle se trouve cette dernière, doit être remis à *null* lors de sa démotion en espace partagé. Contrairement à LRU-PRETI ou FIFO-PRETI, les premiers blocs d'une tâche selon l'ordre global ne sont pas nécessairement ceux de son espace privé. Après un *flop* local, un bloc démis en espace partagé pourrait se trouver précéder ceux de son ancien espace privé du point de vue global. La position physique d'un bloc, la ligne dans laquelle il se trouve, impacte en effet sa position logique.

En cas de défaut de cache, si la tâche provoquant ce défaut ne dispose pas d'un espace privé ou si ce dernier n'a pas encore atteint sa taille limite, le premier bloc de l'espace partagé pour lequel le bit RU n'est pas positionné est évincé. Encore une fois, les blocs sont ici ordonnés selon l'ordre utilisé par l'implémentation pour

parcourir les lignes d'un ensemble.

Si la partition de la tâche provoquant le défaut a atteint sa capacité, la démotion d'un bloc de l'espace privé en espace partagé est requise. Elle correspond à l'éviction du bloc dans le cache privé équivalent à la partition. Dans un cache privé, la ligne de ce bloc évincé est utilisée pour le bloc inséré et participe donc à définir sa position logique au cours d'accès ultérieurs.

Pour maintenir une équivalence comportementale, dans le cas de PRETI, les candidats pour la politique d'éviction dépendent de la position physique du bloc démis. Ils sont compris entre les lignes suivants et précédents le bloc dému dans l'espace privé. Le premier bloc de l'espace partagé compris entre ces deux jalons, à défaut le début ou la fin de l'ensemble, et dont le bit RU n'est pas positionné est choisi pour éviction. Il faut noter que le bloc dému, le premier bloc de l'espace privé donc le RU n'est pas positionné, est un candidat valide le cas échéant.

PLRU-PRETI

La politique de remplacement PLRU est une implémentation approximée d'une politique de remplacement LRU. Pour assurer l'ordre entre les différents blocs dans le cache, les lignes d'un ensemble sont réparties sur les feuilles d'un arbre binaire. Chaque nœud de l'arbre comprend un bit positionné pour désigner celui de ses fils abritant le bloc le plus ancien. Lorsqu'une éviction est requise, une descente le long de cet arbre, en suivant la direction donnée par chaque nœud, permet de trouver le bloc le plus ancien. Insertion et promotion inversent simplement les bits sur le chemin entre le bloc ciblé et la racine.

En cas d'insertion ou de promotion dans un cache partitionné, la direction indiquée par un nœud ne peut être inversée si ses deux fils abritent les blocs d'un espace privé concurrent à la tâche émettant la requête. Dans le cas contraire, cette dernière pourrait altérer l'ordre des blocs d'espaces privés concurrents. Du point de vue de la tâche émettant la requête toutefois, ne pas effectuer cette inversion implique une divergence comportementale par rapport à un cache privé utilisant la politique de remplacement PLRU.

Pour assurer l'isolation d'espaces privés concurrents, une solution est de restreindre les lignes d'une même partition à un sous-arbre dédié exclusivement à cette partition. Par exemple, l'implémentation proposée dans [49] permet d'identifier le sous-arbre où se situe une partition. Les tailles de partition doivent être exprimées sous la forme de puissances de 2. Les mises à jours affectant l'ordre entre différents blocs peuvent être restreintes au sous-arbre appartenant à la partition ciblée. Une

telle implémentation assure l'équivalence du comportement d'une partition et d'un cache privé de taille équivalente.

Toutefois, elle supporte mal l'allocation et la libération dynamiques d'espaces privés tels que mis en œuvre dans la politique de remplacement PRETI. En effet, dans certaines configurations, il peut être impossible de trouver un sous-arbre libre comprenant un nombre suffisant de lignes appartenant à l'espace partagé pour accommoder la partition d'une nouvelle tâche. De plus, une fois l'occupation d'un sous-arbre débutée par une partition rien ne permet dans le mécanisme proposé d'en incorporer les lignes non encore utilisées dans un espace commun. Cette implémentation convient donc à un partitionnement strict, où la taille des partitions est figée pour la durée de vie du système.

Glossaire

AB *aggressive bypass* : heuristique de bypass agressive. 44, 57

AH *always-hit* : CHMC succès garanti. 29, 40

AM *always-miss* : CHMC défaut garanti. 29, 40, 46

bypass court-circuitage du cache. 3, 22, 26, 41

CAC *cache access classification* : classification d'accès au cache. 33, 34, 44

A *always* : CAC toujours accédé. 33

cache antémémoire. 2, 6

N *never* : CAC jamais accédé. 33

U *uncertain* : CAC accès incertain. 33

UN *uncertain-never* : CAC accès incertain sur premiers accès. 33

CB *conservative bypass* : heuristique de bypass conservative. 43, 57

CCN *cache conflicts number* : nombre de conflits inter-tâches en cache. 38

CFG *control flow graph* : graphe de flot de contrôle. 10, 43

CHMC *cache hit/miss classification* : classification succès ou défaut vis-à-vis du cache. 29, 34, 44

CRPD *cache related preemption delay* : délai de préemption lié au cache. 18

FIFO *first-in first-out* : premier-entré premier-sorti. 90

FM *first-miss* : CHMC défauts sur premiers accès. 29, 40, 48

IB *indeterministic bypass* : heuristique de bypass *non déterministe*. 44, 57

ILP *integer linear programming* : programmation linéaire par nombres entiers. 10, 34

IPC *instructions per cycle* : nombre d'instructions par cycle. 82

IPET *implicit path enumeration technique* : technique d'énumération implicite des chemins. 10, 34, 46

LRU *least recently used* : moins récemment utilisé. 14, 16, 45

MRU *most recently used* : plus récemment utilisé. 15

NC *not-classified* : CHMC non classifiée. 29, 46

NP-EDF *non-preemptive earliest deadline first* : non-préemptif, priorité à l'échéance la plus courte. 77

NRU *not recently used* : non utilisé récemment. 90

PLRU *pseudo*-LRU : approximation du LRU. 90

PRETI *Partitionned real-time* : partitionnement pour environnement temps-réel. 3, 66, 96

WCET *worst case execution time* : Temps d'exécution au pire cas. 1, 5, 45, 73

locking verrouillage de contenu de cache. 20

Publications de l'auteur

[1] Benjamin Lesage, Isabelle Puaut et André Seznec, « PRETI : Partitionned REal-TIme shaed cache for mixed-criticality real-time systems. », *in 20th International Conference on Real-Time and Network Systems*, nov. 2012.

[2] Damien Hardy, Benjamin Lesage et Isabelle Puaut, « Scalable Fixed-Point Free Instruction Cache Analysis », *in Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*, RTSS '11, p. 204–213, 2011.

[3] Benjamin Lesage, Damien Hardy et Isabelle Puaut, « Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. », *in 18th International Conference on Real-Time and Network Systems*, nov. 2010.

[4] Benjamin Lesage, Damien Hardy et Isabelle Puaut, « WCET Analysis of Multi-Level Set-Associative Data Caches. », *in WCET*, vol. 10 *in OASICS*, 2009.

[5] Maxime Dénès, Benjamin Lesage, Yves Bertot et Adrien Richard, « Formal proof of theorems on genetic regulatory networks », *in SYNASC'09*, 2009.

Bibliographie

[1] *JPL Institutional Coding Standard for the C Programming Language*, rap. tech., Jet Propulsion Laboratory, mars 2009.

[2] A. V. Aho, R. Sethi et J. D. Ullman, *Compilers principles, techniques, and tools*, Addison-Wesley, 1986.

[3] A. Andrei, P. Eles, Z. Peng et J. Rosen, *Predictable implementation of real-time applications on multiprocessor systems-on-chip*, *in* 21st International Conference on VLSI Design, 2008.

[4] J. Archibald et J.-L. Baer, *Cache coherence protocols : evaluation using a multiprocessor simulation model*, ACM Trans. Comput. Syst., 4 (1986).

[5] R. R. Atkinson et E. M. McCreight, *The dragon processor*, SIGARCH Comput. Archit. News, 15 (1987), p. 65–69.

[6] G. Balakrishnan et T. Reps, *Analyzing memory accesses in x86 executables*, *in* Compiler Construction, Springer, 2004, p. 2732–2733.

[7] C. Ballabriga et H. Casse, *Improving the first-miss computation in set-associative instruction caches*, *in* ECRTS '08. Euromicro Conference on Real-Time Systems, july 2008, p. 341 –350.

[8] L. A. Belady, *A study of replacement algorithms for a virtual-storage computer*, IBM Syst. J., 5 (1966), p. 78–101.

[9] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen et C. Bastoul, *The polyhedral model is more widely applicable than you think*, *in* Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction, p. 283–303.

[10] B. Berna et I. Puaut, *PDPA : period driven task and cache partitioning algorithm for multi-core systems.*, *in* RTNS, ACM, 2012, p. 181–189.

[11] G. Bernat, A. Colin et S. Petters, *WCET analysis of probabilistic hard real-time systems*, *in* 23rd IEEE Real-Time Systems Symposium, 2002, p. 279 – 288.

[12] A. Betts et G. Bernat, *Tree-based wcet analysis on instrumentation point graphs*, *in* Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2006., april 2006, p 8 pp.

[13] A. Binstock, *Multi-core processor architecture explained*, 2008.

[14] C. Burguière, *Modéliser la prédiction de branchement pour le calcul de temps d'exécution pire-cas*, Thèse doctorat, Université Paul Sabatier - Toulouse 3, June 2008.

[15] M. Campoy, A. P. Ivars et J. V. B. Mataix, *Static use of locking caches in multitask preemptive real-time systems*, *in* In Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium), 2001.

[16] S. Chattopadhyay et A. Roychoudhury, *Unified cache modeling for wcet analysis and layout optimizations*, *in* 30th IEEE Real-Time Systems Symposium, RTSS 2009, dec. 2009, p. 47 –56.

[17] A. Colin et S. Petters, *Experimental evaluation of code properties for wcet analysis*, *in* 24th IEEE Real-Time Systems Symposium, 2003., dec., p. 190 – 199.

[18] A. Colin et I. Puaut, *A modular & retargetable framework for tree-based wcet analysis*, *in* ECRTS '01 : Proceedings of the 13th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2001, p 37.

[19] P. Cousot et R. Cousot, *Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, *in* POPL '77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, 1977, p. 238–252.

[20] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones et F. J. Cazorla, *Measurement-based probabilistic timing analysis for multi-path programs.*, *in* ECRTS, IEEE Computer Society, 2012, p. 91–101.

[21] D. Eklov, N. Nikoleris, D. Black-Schaffer et E. Hagersten, *Cache pirating : Measuring the curse of the shared cache*, *in* Proceedings of the 2011 International Conference on Parallel Processing, ICPP '11, p. 165–175.

[22] J. Engblom, *Processor Pipelines and Static Worst-Case Execution Time Analysis*, Thèse doctorat, Uppsala University, Computer Systems, 2002.

[23] J. Engblom et A. Ermedahl, *Modeling complex flows for worst-case execution time analysis*, *in* The 21st IEEE Real-Time Systems Symposium, 2000, p. 163 –174.

[24] H. Falk, P. Lokuciejewski et H. Theiling, *Design of a WCET-Aware C Compiler*, *in* Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia, p. 121–126.

[25] C. Ferdinand et R. Wilhelm, *On predicting data cache behavior for real-time systems*, *in* LCTES '98 : Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, Springer-Verlag, 1998, p. 16–30.

[26] C. Ferdinand et R. Wilhelm, *Efficient and precise cache behavior prediction for real-timesystems*, Real-Time Syst., 17 (1999), p. 131–181.

[27] L. Finot, *Milinda-pañha : les questions de Milinda*, Connaissance de l'Orient, Format poche, Gallimard, 1992.

[28] J. Gait, *A probe effect in concurrent programs*, Softw. Pract. Exper., 16 (1986), p. 225–233.

[29] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Professional, 1989.

[30] D. Grund et J. Reineke, *Toward Precise PLRU Cache Analysis*, *in* WCET'10 : 10th International Workshop on Worst-Case Execution Time Analysis.

[31] D. Grund et J. Reineke, *Precise and efficient fifo-replacement analysis based on static phase detection*, *in* ECRTS, 2010.

[32] N. Guan, M. Lv, W. Yi et G. Yu, *WCET Analysis with MRU Caches : Challenging LRU for Predictability*, *in* IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012, p. 55 –64.

[33] D. GULICK, A. LAMBRECHT, M. WEBB, L. HEWITT et B. BARNES, *Programmable bus arbiter including real time priority indicator fields for arbitration priority selection*, déc. 19 1996. WO Patent WO/1996/041,271.

[34] J. Gustafsson, A. Betts, A. Ermedahl et B. Lisper, *The mälardalen WCET benchmarks - past, present and future*, *in* Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis, July 2010.

[35] S. Hahn et D. Grund, *Relational cache analysis for static timing analysis*, *in* 24th Euromicro Conference on Real-Time Systems (ECRTS), july 2012, p. 102 –111.

[36] D. Hardy, B. Lesage et I. Puaut, *Scalable fixed-point free instruction cache analysis*, *in* Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium, p. 204–213.

[37] D. Hardy, T. Piquet et I. Puaut, *Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches*, *in* RTSS'09 : 30th IEEE Real-Time Systems Symposium, dec. 2009, p. 68 –77.

[38] D. Hardy et I. Puaut, *Predictable code and data paging for real time systems*, *in* ECRTS '08 : Proceedings of the 2008 Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2008, p. 266–275.

[39] D. Hardy et I. Puaut, *WCET analysis of multi-level non-inclusive set-associative instruction caches*, *in* RTSS '08 : Proceedings of the 2008 Real-Time Systems Symposium, IEEE Computer Society, 2008, p. 456–466.

[40] D. Hardy et I. Puaut, *Wcet analysis of instruction cache hierarchies*, Journal of Systems Architecture, 57 (2011), p. 677 – 694.

[41] M. D. Hill, *Aspects of cache memory and instruction*, rap. tech., Berkeley, CA, USA, 1987.

[42] N. Holsti, T. Långbacka et S. Saarinen, *Using a worst-case execution time tool for real-time verification of the debie software*, *in* Proceedings of the DASIA 2000 (Data Systems in Aerospace) Conference, 2000.

[43] B. K. Huynh, L. Ju et A. Roychoudhury, *Scope-aware data cache analysis for wcet estimation*, *in* Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE, april 2011, p. 203 –212.

[44] IBM, *PowerPC 403 GA user's manual*, no 253669-033US.

[45] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, no 253669-033US, December 2009.

[46] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr. et J. Emer, *Adaptive insertion policies for managing shared caches*, *in* Proceedings of the 17th international conference on Parallel architectures and compilation techniques, p. 208–219.

[47] K. Jeffay, D. Stanat et C. Martel, *On non-preemptive scheduling of period and sporadic tasks*, *in* Proceedings of Real-Time Systems Symposium (RTSS), dec 1991.

[48] D. Kaseridis, M. F. Iqbal, J. Stuecheli et L. K. John, *MCFQ : Leveraging Memory-level Parallelism and Application's Cache Friendliness for Efficient Management of Quasi-partitioned Last-level Caches*, *in* PACT, 2011.

[49] K. Kedzierski, M. Moreto, F. Cazorla et M. Valero, *Adapting cache partitioning algorithms to pseudo-lru replacement policies*, *in* Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, p. 1 –12.

[50] S. Kim, D. Chandra et Y. Solihin, *Fair cache sharing and partitioning in a chip multiprocessor architecture*, *in* Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04, Washington, DC, USA, IEEE Computer Society, p. 111–122.

[51] D. B. Kirk, *SMART (strategic memory allocation for real-time) cache design*, *in* IEEE Real-Time Systems Symposium, 1989, p. 229–239.

[52] D. L. Kuck, *Structure of Computers and Computations*, John Wiley & Sons, Inc., New York, NY, USA, 1978.

[53] W. Landi et B. G. Ryder, *Pointer-induced aliasing : a problem classification*, *in* Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '91, p. 93–103.

[54] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee et C. S. Kim, *Analysis of cache-related preemption delay in fixed-priority preemptive scheduling*, IEEE Trans. Comput., 47 (1998), p. 700–713.

[55] B. Lesage, D. Hardy et I. Puaut, *WCET analysis of multi-level set-associative data caches*, *in* 9th Int' Workshop on Worst-Case Execution Time Analysis, 2009.

[56] ——, *Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures.*, *in* 18th International Conference on Real-Time and Network Systems, 2010.

[57] B. Lesage, I. Puaut et A. Seznec, *PRETI : Partitionned REal-TIme shaed cache for mixed-criticality real-time systems.*, *in* 20th International Conference on Real-Time and Network Systems, nov. 2012.

[58] Y. Li, V. Suhendra, Y. Liang, T. Mitra et A. Roychoudhury, *Timing analysis of concurrent programs running on shared cache multi-cores*, *in* RTSS'09 : 30th IEEE Real-Time Systems Symposium, dec. 2009, p. 57 –67.

[59] Y.-T. S. Li et S. Malik, *Performance analysis of embedded software using implicit path enumeration*, *in* DAC '95 : Proceedings of the 32nd ACM/IEEE conference on Design automation, ACM, 1995, p. 456–461.

[60] J. Liedtke, H. Haertig et M. Hohmuth, *OS-controlled cache predictability for real-time systems*, *in* RTAS '97 : Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium, Washington, DC, USA, 1997, IEEE Computer Society, p 213.

[61] M. Livani, J. Kaiser et W. Jia, *Scheduling hard and soft real-time communication in the controller area network*, Control Engineering Practice, 7 (1999), p. 1515–1523.

[62] P. Lokuciejewski, H. Falk et P. Marwedel, *WCET-driven Cache-based Procedure Positioning Optimizations*, *in* ECRTS '08. Euromicro Conference on Real-Time Systems, july 2008, p. 321 –330.

[63] T. Lundqvist et P. Stenström, *A method to improve the estimated worst-case performance of data caching*, *in* RTCSA '99 : Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, IEEE Computer Society, 1999, p 255.

[64] T. Lundqvist et P. Stenström, *Timing anomalies in dynamically scheduled microprocessors*, *in* RTSS '99 : Proceedings of the 20th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1999, p 12.

[65] W. Lunniss, S. Altmeyer et R. I. Davis, *Optimising task layout to increase schedulability via reduced cache related pre-emption delays*, *in* Proceedings of the 20th International Conference on Real-Time and Network Systems, p. 161–170.

[66] D. Molka, D. Hackenberg, R. Schone et M. S. Muller, *Memory performance and cache coherency effects on an intel nehalem multiprocessor system*, *in* Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, p. 261–270.

[67] S. S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann Publishers Inc., 1997.

[68] F. Mueller, *Compiler support for software-based cache partitioning*, SIGPLAN Not., 30 (1995), p. 125–133.

[69] F. Mueller, *Static cache simulation and its applications*, Thèse doctorat, 1995.

[70] F. Mueller, *Timing predictions for multi-level caches*, *in* In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems, 1997, p. 29–36.

[71] F. Mueller, *Timing analysis for instruction caches*, Real-Time Syst., 18 (2000), p. 217–247.

[72] H. S. Negi, T. Mitra et A. Roychoudhury, *Accurate estimation of cache-related preemption delay*, *in* CODES+ISSS '03 : Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM, 2003, p. 201–206.

[73] K. J. Nesbit, J. Laudon et J. E. Smith, *Virtual private caches*, SIGARCH Comput. Archit. News, 35 (2007), p. 57–68.

[74] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat et M. Valero, *Hardware support for wcet analysis of hard real-time multicore systems*, *in* Proceedings of the 36th annual international symposium on Computer architecture, ISCA '09, p. 57–68.

[75] M. Paolieri, E. Quiñones, F. Cazorla, R. Davis et M. Valero, *IA3 : An Interference Aware Allocation Algorithm for Multicore Hard Real-Time Systems*, *in* 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011, p. 280 –290.

[76] T. Piquet, O. Rochecouste et A. Seznec, *Exploiting single-usage for effective memory management*, *in* Proceedings of the 12th Asia-Pacific conference on Advances in Computer Systems Architecture, ACSAC '07, p. 90–101.

[77] I. Puaut, *WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems*, *in* ECRTS '06 : Proceedings of the 18th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2006, p. 217–226.

[78] I. Puaut et D. Decotigny, *Low-complexity algorithms for static cache locking in multitasking hard real-time systems*, *in* RTSS '02 : Proceedings of the 23rd IEEE Real-Time Systems Symposium, IEEE Computer Society, 2002, p 114.

[79] P. Puschner et C. Koza, *Calculating the maximum, execution time of real-time programs*, Real-Time Syst., 1 (1989), p. 159–176.

[80] M. K. Qureshi et Y. N. Patt, *Utility-based cache partitioning : A low-overhead, high-performance, runtime mechanism to partition shared caches*, *in* Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006.

[81] J. Reineke, D. Grund, C. Berg et R. Wilhelm, *Timing predictability of cache replacement policies*, Real-Time Syst., 37 (2007), p. 99–122.

[82] A. H. J. Sale, *The basic principles of well-structured code*, Australian Computer Journal, 7 (1975), p. 116–126.

[83] E. Salminen, V. Lahtinen, K. Kuusilinna et T. Hamalainen, *Overview of bus-based system-on-chip interconnections*, *in* ISCAS'02 : IEEE Internation Symposium on Circuits and Systems, 2002.

[84] J. E. Sasinowski et J. K. Strosnider, *A dynamic programming algorithm for cache memory partitioning for real-time systems*, IEEE Trans. Comput., 42 (1993), p. 997–1001.

[85] A. Schrijver, *Theory of linear and integer programming*, John Wiley & Sons, Inc., 1986.

[86] R. Sen et Y. N. Srikant, *Executable analysis using abstract interpretation with circular linear progressions*, *in* Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE '07, p. 39–48.

[87] R. Sen et Y. N. Srikant, *WCET estimation for executables in the presence of data caches*, *in* EMSOFT '07 : Proceedings of the 7th ACM & IEEE international conference on Embedded software, ACM, 2007, p. 203–212.

[88] A. J. Smith, *Cache memories*, ACM Comput. Surv., 14 (1982), p. 473–530.

[89] T. Sondag et H. Rajan, *A More Precise Abstract Domain for Multi-level Caches for Tighter WCET Analysis*, *in* Proceedings of the 2010 31st IEEE Real-Time Systems Symposium, Washington, DC, USA, IEEE Computer Society, p. 395–404.

[90] J. Staschulat, S. Schliecker et R. Ernst, *Scheduling analysis of real-time systems with precise modeling of cache related preemption delay*, *in* ECRTS '05 : Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2005, p. 41–48.

[91] V. Suhendra et T. Mitra, *Exploring locking & partitioning for predictable shared caches on multi-cores*, *in* DAC '08 : Proceedings of the 45th annual conference on Design automation, ACM, 2008, p. 300–303.

[92] Y. Tan, *A prioritized cache for multi-tasking real-time systems*, *in* 11th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), 2003.

[93] C. P. Thacker et L. C. Stewart, *Firefly : a multiprocessor workstation*, SIGARCH Comput. Archit. News, 15 (1987), p. 164–172.

[94] H. Theiling, C. Ferdinand et R. Wilhelm, *Fast and precise WCET prediction by separated cache and path analyses*, Real-Time Syst., 18 (2000), p. 157–179.

[95] A. M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proc. London Math. Soc., 2 (1936), p. 230–265.

[96] A. Valmari, *The state explosion problem*, *in* Lectures on Petri Nets I : Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, Springer-Verlag, 1998, p. 429–528.

[97] X. Vera, B. Lisper et J. Xue, *Data cache locking for higher program predictability*, *in* SIGMETRICS '03 : Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems, ACM, 2003, p 272.

[98] ——, *Data caches in multitasking hard real-time systems*, *in* RTSS '03 : Proceedings of the 24th IEEE International Real-Time Systems Symposium, IEEE Computer Society, 2003, p 154.

[99] X. Vera, B. Lisper et J. Xue, *Data cache locking for tight timing calculations*, ACM Trans. Embed. Comput. Syst., 7 (2007), p. 1–38.

[100] X. Vera et J. Xue, *Let's study whole-program cache behaviour analytically*, *in* HPCA '02 : Proceedings of the 8th International Symposium on High-Performance Computer Architecture, IEEE Computer Society, 2002, p 175.

[101] J. Wegener et M. Grochtmann, *Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing*, Real-Time Syst., 15 (1998), p. 275–298.

[102] J. Wegener et F. Mueller, *A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints*, Real-Time Syst., 21 (2001), p. 241–268.

[103] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller et M. G. Harmon, *Timing analysis for data caches and set-associative caches*, *in* Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97), Washington, DC, USA, 1997, IEEE Computer Society, p. 192–.

[104] R. T. White, F. Mueller, C. Healy, D. Whalley et M. Harmon, *Timing analysis for data and wrap-around fill caches*, Real-Time Syst., 17 (1999), p. 209–233.

[105] N. Williams, B. Marre, P. Mouy et M. Roger, *PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis*, vol. 3463/2005 de Lecture Notes in Computer Science, Springer Berlin, March 2005, p. 281–292.

[106] M. E. Wolf et M. S. Lam, *A data locality optimizing algorithm*, *in* PLDI '91 : Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, ACM, 1991, p. 30–44.

[107] Y. Xie et G. H. Loh, *PIPP : Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches*, *in* 36th Intl. Symp. on Computer Architecture, 2009.

[108] J. Yan et W. Zhang, *WCET analysis for multi-core processors with shared L2 instruction caches*, *in* RTAS '08 : Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, 2008.

Résumé

Les tâches critiques en systèmes temps-réel sont soumises à des contraintes temporelles et de correction. La validation d'un tel système repose sur l'estimation du comportement temporel au pire cas de ses tâches. Le partage de ressources, inhérent aux architectures multi-cœurs, entrave le calcul de ces estimations. Le comportement temporel d'une tâche dépend de ses rivales du fait de l'arbitrage de l'accès aux ressources ou de modifications concurrentes de leur état.

Cette étude vise à l'estimation de la contribution temporelle de la hiérarchie mémoire au pire temps d'exécution de tâches critiques. Les méthodes existantes, pour caches d'instructions, sont étendues afin de supporter caches de données privés et partagés, et permettre l'analyse de hiérarchies mémoires riches. Le court-circuitage de cache est ensuite utilisé pour réduire la pression sur les caches partagés. Nous proposons à cette fin différentes heuristiques basées sur la capture de la réutilisation de blocs de cache entre différents accès mémoire. Notre seconde proposition est la politique de partitionnement Preti qui permet l'allocation d'un espace sans conflits à une tâche. Preti favorise aussi les performances de tâches non critiques concurrentes aux temps-réel dans les systèmes de criticité hybride.

Abstract

Critical tasks in the context of real-time systems submit to both timing and correctness constraints. Whence, the validation of a real-time system rely on the estimation of its tasks' *Worst case execution times*. Resource sharing, as it occurs on multicore architectures, hinders the computation of such estimates. The timing behaviour of a task is impacted by its concurrents, whether because of resource access arbitration or concurrent modifications of a resource state.

This study focuses on estimating the contribution of the memory hierarchy to tasks' worst case execution time. Existing analysis methods, defined for instruction caches, are extended to support private and shared data caches, hence allowing for the analysis of rich memory hierarchies. Cache bypass is then used to reduce the pressure laid by concurrent tasks on shared caches levels. We propose different bypass heuristics, based on the capture of cache blocks' reuse between memory accesses. Our second proposal is the Preti partitioning scheme which allows for the allocation to tasks of a cache space, free from inter-task conflicts. Preti offers the added benefit of providing for average-case performance to non-critical tasks concurrent to real-time ones on hybrid criticality systems.