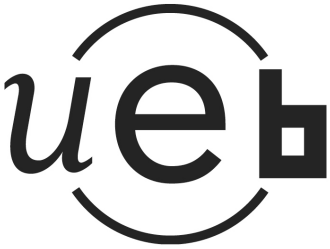
Analyse pire cas pour processeur multi-cœurs disposant de caches partagés

Damien Hardy

N° d'ordre : 4244 **ANNÉE 2010**



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

Damien Hardy

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

Analyse pire cas pour processeur multi-cœurs disposant de caches partagés

Thèse soutenue à Rennes le 9 décembre 2010

devant le jury composé de :

Olivier Ridoux
Professeur à l'Université de Rennes 1
président

Gilles Grimaud
Professeur à l'Université des Sciences et Technologies de Lille
rapporteur

Pascal Sainrat
Professeur à l'Université Paul Sabatier, Toulouse
rapporteur

Isabelle Puaut
Professeur à l'Université de Rennes 1
directrice de thèse

membre invité :

André Seznec
Directeur de recherche à l'INRIA-Rennes

Ce n'est pas le temps qui passe, mais nous qui le traversons.
proverbe

Remerciements

Je tiens en premier lieu à remercier Olivier Ridoux, Professeur à l'Université de Rennes 1, qui m'a fait l'honneur de présider le jury de ma thèse.

Je remercie chaleureusement Gilles Grimaud, Professeur à l'Université des Sciences et Technologies de Lille, et Pascal Sainrat, Professeur à l'Université Paul Sabatier de Toulouse, d'avoir accepté de rapporter cette thèse ainsi que Philippe Clauss, Professeur à l'Université de Strasbourg, d'avoir accepté d'être membre de mon jury.

Je tiens à exprimer toute ma gratitude envers Isabelle Puaut pour avoir dirigé mes travaux de thèse. Je salue sa disponibilité, son écoute, ses conseils avisés ainsi que sa volonté de me transmettre une certaine vision de la recherche.

Je tiens également à remercier respectueusement André Seznec, responsable de l'équipe CAPS puis ALF, de m'avoir accueilli mais également pour sa participation à mon jury ainsi que pour le regard objectif qu'il a su porter sur mes travaux.

Je remercie sincèrement l'ensemble des membres présents et passés des équipes ALF et CAPS tant pour les discussions scientifiques enrichissantes que celles plus divertissantes lors des pauses. La convivialité et la bonne humeur de chacun ont contribué à rendre mes années de thèse très agréables.

Une pensée pour toutes les personnes dont j'ai croisé le chemin au cours de ces dernières années et qui m'ont encouragé d'une manière ou d'une autre.

Je tiens finalement à remercier mes amis proches, ma famille, ma sœur Chloé et surtout mes parents pour leur soutien et leur patience. Cette thèse, je la dois à tous ceux qui ont su m'apporter leur aide quand j'en avais besoin.
Merci.

Table des matières

Introduction

Depuis les premiers ordinateurs personnels du début des années quatre-vingt, les systèmes informatisés se sont largement diversifiés et font maintenant partie intégrante de notre quotidien. Nous les utilisons dans le cadre de notre travail, de nos loisirs mais, également pour améliorer notre confort et notre sécurité. Nous les retrouvons par exemple dans nos ordinateurs personnels, dans nos téléphones portables, dans nos appareils multimédia, dans nos véhicules...

Entre les différents systèmes informatisés, nous pouvons distinguer les systèmes temps réel qui exécutent des applications interagissant avec l'environnement dans lequel ils se situent. La communication avec l'environnement est effectuée d'une part à l'aide de capteurs servant à collecter des informations sur l'environnement. Ces informations sont ensuite traitées par les calculateurs du système déterminant les actions à réaliser par les actionneurs en fonction des événements générés par l'environnement.

Ces systèmes sont de plus en plus utilisés dans de nombreux domaines comme l'avionique, les centrales nucléaires ou encore l'automobile. De tels systèmes doivent bien sûr fournir des résultats valides en réponse aux événements de l'environnement mais ils doivent en plus fournir ces résultats en un temps déterminé afin de respecter les contraintes temporelles du système dont la plus courante est l'échéance de terminaison au plus tard. Le respect des contraintes temporelles est plus ou moins important suivant que l'on se place dans le cadre du temps réel *souple* ou *strict*. Pour les systèmes temps-réel souple, les contraintes sont définies pour assurer une certaine qualité de service et le non-respect occasionnel n'entraîne pas de problèmes importants. Par contre, le respect des contraintes de temps est impératif dans les systèmes temps-réel strict pour en assurer le bon fonctionnement. Le non respect des contraintes temporelles peut dans de tels systèmes entraîner des conséquences économiques, écologiques, humaines catastrophiques.

Pour valider le bon fonctionnement des systèmes temps-réel, il est donc impératif de garantir la terminaison au plus tard à l'échéance de chacune des tâches s'exécutant sur le système. Cette validation repose sur une connaissance du temps d'exécution pire cas de chacune des tâches s'exécutant sur le système et pouvant être déterminé soit par mesures, soit par analyse statique.

Afin d'offrir de plus en plus de fonctionnalités, les systèmes temps-réel se doivent d'être de plus en plus performants. Pour ce faire, ils disposent de mécanismes matériels comme les mémoires cache, permettant de réduire considérablement les latences d'accès à la mémoire. La présence de ce mécanisme matériel permet d'améliorer les performances du système. Cependant l'obtention du temps d'exécution pire cas devient plus complexe en raison de la gestion dynamique de ce mécanisme amenant une source d'indéterminisme sur les temps d'accès à la mémoire. De plus, sur les architectures multi-cœurs actuelles, ces mémoires sont organisées de façon hiérarchique et certains niveaux peuvent être partagés entre différentes applications s'exécutant simultanément, ce qui introduit des conflits supplémentaires lors de l'exécution et amène donc une source d'indéterminisme supplémentaire pour l'obtention du temps d'exécution pire cas.

Questions de recherche

Les questions de recherche auxquelles nous essaierons de répondre dans la suite de ce document, vis-à-vis de l'estimation du pire temps d'exécution utilisée lors du processus de validation des systèmes temps-réel strict, sont les suivantes :

Q1. Est-il possible d'analyser de façon sûre et précise le comportement temporel pire cas des accès mémoire effectués au sein d'une hiérarchie de mémoires cache ?

Q2. Est-il possible d'estimer de façon sûre et précise le pire temps d'exécution de tâches exécutées sur des processeurs multi-cœurs disposant de caches partagés ?

Contributions

Dans ce document, nous nous intéressons aux effets du comportement temporel pire cas des hiérarchies de mémoires cache sur le pire temps d'exécution. Afin de déterminer ces effets, il nous faut déterminer le comportement pire cas de chaque niveau de cache composant une hiérarchie ainsi que les interactions entre ces différents niveaux. Afin de garantir la sûreté de l'estimation du pire temps d'exécution, nous utilisons des méthodes d'analyse statique pour déterminer le comportement temporel pire cas de ces mémoires.

Notre première contribution, répondant à la première question, se focalise sur une méthode d'analyse statique, permettant de déterminer le comportement temporel pire cas de chaque niveau de cache ainsi que l'ensemble des interactions possibles entre ces différents niveaux. L'objectif de cette méthode est de déterminer le comportement temporel pire cas de chaque accès se produisant au sein de la hiérarchie mémoire. Les politiques de gestion de hiérarchie non-inclusive,

inclusive et exclusive sont prises en compte par l'analyse proposée ainsi que les politiques de remplacement de cache les plus courantes. La notion clé permettant de garantir la sûreté de l'analyse, vis-à-vis de l'estimation du comportement temporel pire cas, repose sur l'introduction d'une nouvelle classification appelée *classification d'accès au cache* permettant de modéliser et prendre en compte, lors de l'analyse, les interactions entre les niveaux.

La seconde contribution, quant à elle, se concentre sur la prise en compte lors de l'analyse des niveaux de cache partagés, présent dans la plupart des processeurs multi-cœurs actuels, afin de répondre à la seconde question. Nous étendons dans un premier temps l'analyse précédente, afin de déterminer et de prendre en compte les conflits occasionnés dans les niveaux partagés par l'ensemble des tâches du système. Bien que cette approche soit sûre, nous verrons que seule, elle conduit rapidement l'estimation du temps d'exécution pire cas d'une tâche à ne pouvoir détecter aucune réutilisation dans ces niveaux. Nous proposerons, dans un second temps, une méthode basée sur un mécanisme de *bypass* des lignes de cache à usage unique, afin de réduire considérablement les interférences dans ces niveaux et ainsi, capturer de la réutilisation lors de l'analyse, ce qui permet de rendre plus précise l'estimation du pire temps d'exécution.

Notre travail se concentrera essentiellement sur les caches d'instructions. Nous aborderons néanmoins des extensions possibles de chacune de nos méthodes pour analyser et prendre en compte le comportement temporel pire cas des caches de données. Ces extensions, dont certaines sont d'ores et déjà réalisées, ne seront qu'abordées dans ce document car elles ont été réalisées en collaboration avec un étudiant lors de son stage de master.

Organisation du document

La suite de ce document s'organise de la façon suivante. Le premier chapitre présente un état de l'art des méthodes d'estimation du pire temps d'exécution. Dans ce chapitre, nous commencerons par présenter les méthodes d'estimation du temps d'exécution pire cas couramment employées lors de la validation des systèmes temps-réel. Nous rentrerons ensuite plus précisément dans le sujet de ce document, en nous intéressant aux mémoires cache. Nous expliquerons leur fonctionnement et nous verrons les méthodes d'analyse statique existantes ainsi que diverses approches permettant d'améliorer la prévisibilité de ce mécanisme matériel, dans le cadre de l'estimation du comportement temporel pire cas.

Nous présenterons dans le chapitre 2 notre première contribution, à savoir une méthode d'estimation du contenu pire cas des hiérarchies mémoires [40, 42]. Nous commencerons par un exemple montrant que l'idée intuitive consistant à propager de façon systématique les accès incertains dans les niveaux inférieurs

de la hiérarchie, lors de l'analyse, peut mettre en défaut la sûreté de l'estimation du pire temps d'exécution. Nous introduirons ensuite la notion de classification d'accès au cache permettant de modéliser et prendre en compte, de façon sûre, le comportement des accès incertains lors de l'analyse d'un niveau de cache appartenant à une hiérarchie de caches disposant d'une politique de gestion non-inclusive ou inclusive. Nous verrons que les hiérarchies de caches disposant d'une politique de gestion exclusive sont quelques peu différentes mais, peuvent tout de même être analysées en se basant sur les propriétés de ce mode de gestion. Nous proposerons ensuite une solution permettant de modéliser différentes politiques de remplacement de cache lors de l'analyse d'un niveau de cache, indépendamment de la politique de gestion utilisée. Une évaluation expérimentale des différentes analyses proposées sera menée montrant que l'approche permet généralement de rendre plus précise l'estimation du pire temps d'exécution comparativement aux analyses existantes ne considérant qu'un seul niveau de cache. Nous terminerons ce chapitre en abordant des extensions possibles à cette méthode.

Dans le chapitre 3, nous détaillerons notre seconde contribution, à savoir une méthode permettant de prendre en compte lors de l'analyse les niveaux de caches partagés des hiérarchies mémoires telles que nous les trouvons dans les processeurs multi-cœurs. L'approche proposée [38] consiste d'une part à estimer les conflits, dans les niveaux partagés, occasionnés par les tâches s'exécutant simultanément à la tâche analysée et, d'autre part, à les prendre en compte lors de l'estimation du comportement pire cas de chaque accès mémoire. Néanmoins, cette solution seule peut amener à une importante surestimation du pire temps d'exécution. La partie principale de cette contribution repose sur une méthode permettant de réduire de façon importante les conflits. Pour ce faire nous exploiterons un mécanisme matériel permettant d'éviter le stockage, dans les niveaux partagés, des lignes de cache identifiées statiquement comme n'étant jamais réutilisées avant d'être évincées. Nous évaluerons le gain apporté par l'utilisation de ce mécanisme lors de l'estimation du pire temps d'exécution comparativement à une analyse où ce type de mécanisme est absent lors de l'étude expérimentale. Nous conclurons ce chapitre en indiquant des directions possibles pour des travaux futurs.

En conclusion de ce document, nous reprendrons les points importants des différentes contributions et nous discuterons des perspectives de ce travail ainsi que de quelques problèmes ouverts.

Chapitre 1

État de l'art

1.1 Introduction

La validation des systèmes temps-réel, vis-à-vis des contraintes temporelles, se doit de garantir la terminaison au plus tard à l'échéance de chaque tâche composant le système en se basant sur la faisabilité de l'ordonnancement [55, 14, 5, 21, 37]. Cette validation repose sur le pire temps d'exécution (en anglais : *Worst Case Execution Time (WCET)*) de chacune des tâches, celui-ci représentant une borne supérieure de tous les temps d'exécution possibles.

Les systèmes temps-réel se doivent également d'être performants pour offrir de plus en plus de fonctionnalités à l'utilisateur. Pour cela, les architectures exécutant ces systèmes, disposent notamment d'une hiérarchie mémoire. L'intérêt principal de cette hiérarchie mémoire est d'offrir à l'utilisateur le plus grand espace mémoire disponible tout en gardant un temps d'accès le plus rapide possible en se basant sur les principes de localité spatiale et temporelle des applications. Pour ce faire, elle dispose de différents types de mémoires tels que les mémoires cache et la mémoire principale ayant chacun des capacités de stockage et des temps d'accès différents.

La gestion de la hiérarchie mémoire est d'une part transparente pour le programmeur et d'autre part gérée dynamiquement afin d'améliorer le temps moyen d'exécution. Cependant, cette gestion dynamique amène de l'indéterminisme sur les temps d'accès mémoire. En effet, pour deux exécutions différentes, le contenu de chacun des composants de la hiérarchie mémoire est potentiellement différent. Ainsi, prévoir le temps d'accès à une donnée dans la hiérarchie mémoire s'avère complexe car il est fonction de son emplacement dans la hiérarchie lors de l'exécution. Dans les systèmes temps-réel, cette dynamicité se trouve donc être une source de difficultés pour l'estimation du pire temps d'exécution.

Depuis deux décennies, de nombreux travaux de recherche se sont focalisés sur l'estimation du pire temps d'exécution en général et sur le comportement temporel

pire cas de la hiérarchie mémoire en particulier, que nous allons détailler dans la suite de ce chapitre.

1.1.1 Organisation du chapitre

La suite de ce chapitre est organisée de la manière suivante. Nous commencerons par une vue d'ensemble (section 1.2) des méthodes d'estimation du pire temps d'exécution puis, nous étudierons deux méthodes statiques d'estimation existantes (section 1.3). Nous nous intéresserons ensuite aux mémoires cache en présentant leur fonctionnement et les méthodes d'analyse statique permettant d'estimer leur comportement temporel pire cas (section 1.4). Nous étudierons différentes approches proposées dans la littérature pour rendre plus prévisible leur comportement (section 1.5). Nous discuterons de certaines limites dans les méthodes existantes vis-à-vis des architectures actuelles, comme les processeurs multi-cœurs, pour garantir la validation des systèmes s'exécutant sur des architectures modernes, en conclusion de ce chapitre.

1.2 Méthodes d'estimation du temps d'exécution pire cas : vue d'ensemble

Comme nous venons de le voir, la validation des systèmes temps-réel, vis-à-vis des contraintes temporelles, permet de garantir la terminaison au plus tard à l'échéance de chaque tâche composant le système en se basant sur la faisabilité de l'ordonnancement. Cette validation repose sur le pire temps d'exécution (en anglais : *Worst Case Execution Time (WCET)*) de chacune des tâches, celui-ci représentant une borne supérieure de tous les temps d'exécution possibles. Cependant, l'obtention du pire temps d'exécution d'une tâche est dans le cas général un problème indécidable [18, 88] nous amenant de ce fait à estimer une borne supérieure du WCET. L'estimation de cette borne est calculée en nombre de cycles processeurs et afin d'être correcte et la plus exploitable possible elle doit être sûre et précise.

Définition 1.1 (sûreté) *Une borne du WCET d'une tâche est sûre si elle est supérieure à tous les temps d'exécution possibles.*

Cette condition permet de garantir le respect des contraintes de temps. En effet, lors de la phase de validation, il est nécessaire de s'assurer que chaque tâche termine bien son exécution avant son échéance de terminaison au plus tard. Une telle vérification est valide si le temps d'exécution considéré est bien supérieur à tous les temps d'exécution possibles.

Définition 1.2 (précision) *Plus une borne du WCET d'une tâche est proche du maximum de tous les temps d'exécution possibles plus elle est précise.*

Il est important que l'estimation du WCET sur-approxime le moins possible le WCET réel. Une trop grande sur-approximation peut entraîner un échec de faisabilité sur un système donné ou amener les concepteurs à surestimer les ressources matérielles nécessaires amenant à un coût de réalisation plus important.

Afin de simplifier l'analyse d'un système, l'estimation du pire temps d'exécution d'une application temps-réel est généralement effectuée en considérant de façon indépendante chaque tâche la composant. De ce fait, l'estimation du pire temps d'exécution n'est valide que si les tâches sont exécutées en isolation. L'impact du système sur le temps d'exécution de la tâche comme par exemple les interruptions ou encore les délais liés aux préemptions dans le cas de systèmes préemptifs multi-tâches, n'est pas directement intégré dans l'estimation du pire temps d'exécution des tâches. Cette considération est néanmoins réaliste pour des architectures uni-cœur, car les interférences causées par le système d'exploitation peuvent être prises en compte lors des tests d'ordonnançabilité [55, 14, 5, 21, 37] en se basant sur des méthodes de calcul de temps de préemption [71, 98, 97, 84, 16].

Bien que l'obtention d'une borne supérieure du pire temps d'exécution soit un problème simple à énoncer, il n'en reste pas moins un problème complexe à traiter car dans le cas général, un programme exécuté sur une architecture cible ne dispose pas d'un temps d'exécution unique. En effet, le temps d'exécution d'une application peut varier en fonction de différents critères liés d'une part à l'application elle-même et d'autre part aux caractéristiques de l'architecture matérielle sur laquelle elle est exécutée.

Pour de nombreuses applications, les calculs à réaliser lors de l'exécution sont dépendant de ses données d'entrée. En fonction de celles-ci, différents traitements peuvent être réalisés par l'application, amenant à des chemins d'exécution différents et donc des temps d'exécution potentiellement différents. Les méthodes d'estimation de WCET doivent donc être en mesure de prendre en compte cette source de variabilité.

L'autre source de variation du temps d'exécution provient de l'architecture matérielle sur laquelle l'application est exécutée. Les processeurs sont conçus avec comme objectif d'améliorer le temps moyen d'exécution des applications. Pour ce faire, différents mécanismes matériels comme les caches, les *pipelines*, les prédicteurs de branchements... sont maintenant présents dans la plupart des processeurs actuels. Cependant, ces mécanismes gérés dynamiquement afin d'améliorer le temps moyen d'exécution sont sources d'indéterminisme et peuvent pour deux instances d'exécution d'une application disposant du même jeu d'entrée amener à des temps d'exécution différents en fonction de leur état initial. Les méthodes d'estimation de WCET doivent donc prendre en compte le comportement dyna-

mique de ces mécanismes matériels de façon précise afin de ne pas ni sous-estimer ni trop surestimer le pire temps d'exécution.

Ces deux sources de variation ne sont pas décorrelées l'une de l'autre, impliquant que le temps d'exécution d'une tâche est dépendant à la fois du chemin d'exécution emprunté et de l'état initial des mécanismes matériels. Les méthodes d'estimation doivent donc prendre en compte conjointement ces deux sources de variation lors de l'estimation du pire temps d'exécution.

L'estimation du pire temps d'exécution est un problème réputé difficile par la communauté temps-réel qui est étudié depuis plus de vingt ans et a donné lieu à de nombreuses méthodes d'estimation synthétisées dans [79, 106]. Elles peuvent être regroupées dans les deux catégories suivantes : méthodes dynamiques et méthodes statiques.

1.2.1 Classe de méthodes dynamiques

Les méthodes dynamiques consistent à exécuter chacune des tâches, composant une application, sur un système réel ou sur un simulateur avec différents jeux d'entrées et à mesurer le temps d'exécution de chacune des instances. L'estimation du pire temps d'exécution se dérive simplement en retournant le pire temps d'exécution mesuré.

Les jeux d'entrées peuvent être fournis de façon explicite par l'utilisateur ou générés de façon automatique. Cette génération peut utiliser une recherche exhaustive sur la longueur des chemins [108] mais, le temps de génération de tous les jeux de test ainsi que le temps d'évaluation pour chacun d'eux est généralement trop long pour que l'approche soit exploitable en pratique. Un autre type d'approche consiste à utiliser un algorithme génétique [103] ou du recuit simulé afin d'obtenir des jeux de test maximisant le temps d'exécution de l'application.

A l'exception des méthodes dynamiques exhaustives qui sont peu exploitables en pratique, les méthodes dynamiques posent un problème vis-à-vis de la propriété de sûreté requise pour la validation des systèmes. En effet, générer des paramètres d'entrées couvrant le pire chemin d'exécution de l'application s'avère très difficile voire impossible. De ce fait, ces méthodes ne sont pas en mesure de garantir que le plus grand temps d'exécution ainsi mesuré est toujours supérieur à toutes les exécutions possibles. Elles ont cependant l'avantage d'éliminer la phase de modélisation du comportement des mécanismes matériels complexes. Dans le cas de systèmes temps-réel souple, ce type d'approche peut s'avérer suffisante pour garantir la qualité de services des applications.

1.2.2 Classe de méthodes statiques

Les méthodes statiques analysent la structure du programme sans l'exécuter permettant ainsi de fournir une estimation du pire temps d'exécution indépendante du jeu d'entrée et de l'état initial des mécanismes matériels, garantissant ainsi la sûreté de l'estimation du WCET.

Généralement, les méthodes d'analyse statique réalisent un découplage entre l'analyse du comportement pire cas des mécanismes matériels, appelé communément analyse *bas niveau* et l'analyse du flot de contrôle de l'application, appelé communément analyse *haut niveau*. L'analyse bas niveau fournit une estimation du temps d'exécution pire cas d'une portion du programme en se basant sur un modèle d'architecture matérielle, typiquement les caches, les *pipelines*... Le résultat de l'analyse bas niveau est ensuite utilisé par l'analyse haut niveau pour déterminer le pire temps d'exécution de la tâche en se basant sur une représentation de sa structure modélisant les chemins d'exécution possibles.

Dans [2], un découpage physique est même réalisé entre l'analyse bas niveau, effectuée sur la machine cible, et l'analyse haut niveau, effectuée sur un calculateur, afin de garantir d'une part le respect des contraintes de temps mais également la confidentialité d'applications critiques comme celles s'exécutant sur les cartes à puce.

Analyse bas niveau

L'analyse bas niveau permet de déterminer statiquement des informations temporelles sur le pire temps d'exécution d'une séquence d'instructions (bloc de base) en se basant sur une modélisation de l'architecture matérielle. La difficulté de cette analyse est liée à la prise en compte de certains éléments matériels comme les *pipelines* introduisant du parallélisme dans l'exécution des instructions ou encore les caches introduisant une variation du temps d'exécution des instructions.

Ce document étant orienté sur la hiérarchie mémoire, nous détaillerons par la suite différentes méthodes traitant des mémoires cache. Des travaux ont également été proposés pour les *pipelines* mais leur étude se limitera dans ce document aux principes de base et à certaines considérations lors de l'estimation du pire temps d'exécution. D'autres éléments d'architectures ont été étudiés dans le cadre de l'analyse bas niveau comme les mécanismes de pré-chargement [48, 109] ou encore les prédicteurs de branchement [11, 23, 17].

Pour simplifier, considérons pour le moment, dans cette vue d'ensemble, que l'on dispose d'une borne supérieure du pire temps d'exécution de chacun des blocs de base et qu'elle est constante et indépendante du contexte d'exécution.

Analyse haut niveau

Le calcul d'une borne supérieure du WCET s'effectue à partir d'une représentation haut niveau du programme à analyser en utilisant le pire temps d'exécution de chacune des séquences d'instructions déterminé par l'analyse bas niveau. Nous allons présenter dans la section suivante deux approches différentes pour ce calcul : une méthode basée sur l'arbre syntaxique et une méthode basée sur l'énumération implicite des chemins.

1.3 Méthodes d'estimation haut niveau du pire temps d'exécution

Nous détaillons, dans cette partie, deux méthodes d'estimation haut niveau du pire temps d'exécution. Les structures de données manipulées par ces méthodes peuvent être obtenues soit directement à l'intérieur d'un compilateur soit en externe en exploitant le code objet du programme.

1.3.1 Méthode basée sur l'arbre syntaxique

Cette méthode proposée dans [80] utilise une représentation du programme sous la forme d'un arbre syntaxique. Chaque nœud de ce type d'arbre représente une structure de contrôle du langage de haut niveau, comme illustré par la figure 1.1. Par exemple, un nœud SEQ représente une séquence de blocs où chacun des fils peut être une feuille représentant un bloc de base ou une autre structure du langage (boucle, conditionnelle...).

La borne supérieure du WCET est calculée de façon récursive en partant des feuilles de l'arbre syntaxique (contenant la borne supérieure du WCET des blocs de base) et en remontant l'information jusqu'à la racine de l'arbre, celle-ci contenant ainsi une borne supérieure du WCET de la tâche analysée. Pour chaque nœud, nous calculons sa contribution au pire temps d'exécution en maximisant le temps d'exécution de ses fils. Par exemple pour une structure conditionnelle (*if-then-else*), sa contribution sera le pire temps du test (*if*) auquel nous ajoutons le pire temps maximum entre les branches *then* et *else*, pour être indépendant des données d'entrées. Le tableau 1.1 illustre les formules permettant de déterminer la contribution au pire temps d'exécution des principales structures de contrôle.

Nous pouvons remarquer la présence de *maxiter* pour la structure de boucle qui représente le nombre maximum d'itérations. Cette information peut être détectée automatiquement dans certains cas [1] et lorsque c'est impossible en raison de l'impossibilité, en général, de détecter la terminaison de programme [18], elle est ajoutée par le programmeur.

WHILE BB_1 DO
  IF BB_2 THEN
    BB_3
  ELSE
    BB_4
  FI
  BB_5
DONE
BB_6

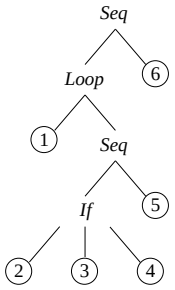


FIG. 1.1: Exemple d'arbre syntaxique obtenu à partir d'un code source. Les nœuds de cet arbre représentent les structures de contrôle du langage de haut niveau. Les fils des nœuds peuvent être soit des feuilles représentant les blocs de base soit d'autres nœuds représentant alors des structures de contrôle imbriquées.

Structure	Formule
$S1; ...; Sn$	$WCET(S_1) + ... + WCET(S_n)$
`if` B `then` S_1 `else` S_2	$WCET(B) + max(WCET(S_1), WCET(S_2))$
`while` B `do` S `done`	$maxiter * (WCET(B) + WCET(S)) + WCET(B)$

TAB. 1.1: WCET des structures de contrôle

Cette méthode fournit un arbre temporel contenant le pire temps d'exécution de chacun des nœuds de l'arbre syntaxique. Cette information est intéressante lors de phases d'optimisation car elle est directement reliée au code source et permet d'identifier les parties coûteuses en temps de calcul. De plus, cette méthode d'estimation étant réduite au parcours d'un arbre ou chaque nœud doit être visité une fois, elle est de ce fait peu coûteuse en terme de temps de calcul et passe donc relativement bien à l'échelle lors de l'analyse de programmes de taille conséquente. Néanmoins, la représentation du programme sous la forme d'un arbre syntaxique du langage de haut niveau peut s'avérer difficile à maintenir lors de certaines optimisations réalisées par le compilateur. En effet, il faut être en mesure d'effectuer la correspondance entre le binaire et le langage de haut niveau ce qui peut se révéler difficile dans le cas d'optimisations tels que le déroulage de boucles ou certaines transformations de code. Finalement, il est difficile avec cette représentation de prendre en compte certaines informations sur le flot d'exécution d'un programme comme notamment des informations sur

les chemins infaisables, c'est à dire les chemins qui ne seront jamais empruntés lors de l'exécution de la tâche. Ces informations permettent pourtant de raffiner l'estimation du pire temps d'exécution.

1.3.2 Méthode d'énumération implicite des chemins

La méthode d'énumération implicite des chemins (en anglais : *Implicit Path Enumeration Techniques (IPET)*) proposée dans [53, 81] repose sur une modélisation de la tâche analysée sous la forme d'un graphe de flot de contrôle.

Un graphe de flot de contrôle est constitué d'un ensemble de nœuds représentant les blocs de base du programme. Un bloc de base est constitué d'une suite d'instructions uniquement séquentielle et possède un seul point d'entrée et de sortie. Les arcs du graphe, eux, modélisent les relations (prédécesseur, successeur) entre les différents blocs de base. Cette représentation, illustrée par la figure 1.2, permet de décrire tous les chemins d'exécution possibles entre les différents blocs de base. Pour l'estimation du WCET, le graphe de flot de contrôle dispose en plus du nombre maximum d'itérations des boucles.

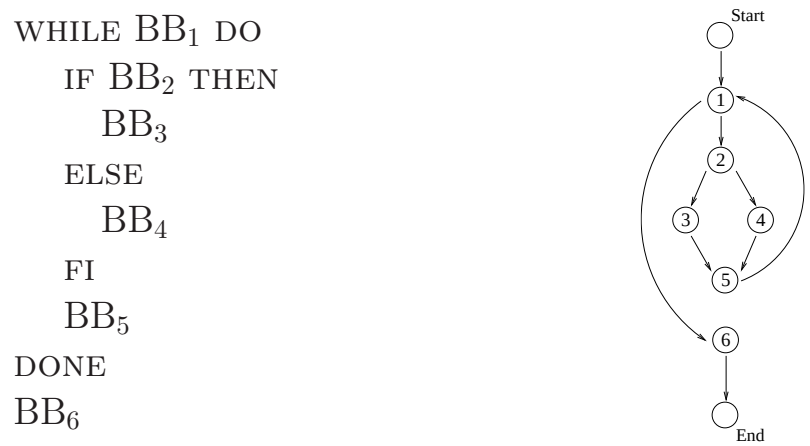


Fig. 1.2: Exemple de graphe de flot de contrôle obtenu à partir d'un code source. Les nœuds de ce graphe représentent les blocs de base et les arcs représentent les flots d'exécution possibles.

La méthode d'énumération implicite des chemins repose sur une transformation du graphe de flot de contrôle en un système de contraintes linéaires et permet ainsi de couvrir tous les chemins possibles sans les énumérer. Ce système décrit la structure du graphe sous la forme de contraintes de flot, où la somme des occurrences des arcs (noté $e_{i,j}$) prédécesseurs d'un nœud est égale à la somme

des occurrences des arcs successeurs de ce nœud et représente le nombre d'occurrence du nœud (noté n_i). Dans le cas de boucles, le nombre d'itérations maximal d'une boucle est intégré dans ce système en indiquant, par exemple, le nombre maximum de fois où les arcs arrières peuvent être empruntés.

$$
\begin{array}{l}
\textit{Contraintes de flot} : \\
\\
n_{Start} = 1 \\
n_{Start} = e_{Start,1} \\
n_1 = e_{Start,1} + e_{5,1} \\
n_1 = e_{1,6} + e_{1,2} \\
n_2 = e_{1,2} \\
n_2 = e_{2,3} + e_{2,4} \\
n_3 = e_{2,3} \\
n_3 = e_{3,5} \\
n_4 = e_{2,4} \\
n_4 = e_{4,5} \\
n_5 = e_{3,5} + e_{4,5} \\
n_5 = e_{5,1} \\
n_6 = e_{1,6} \\
n_6 = e_{6,End} \\
n_{End} = e_{6,End} \\
n_{End} = 1
\end{array}
\qquad
\begin{array}{l}
\textit{Nombre d'itérations maximum} : \\
\\
e_{5,1} \leq maxiter \\
\\
\\
\\
\\
\textit{Fonction objectif} : \\
\\
max(n_1{*}w_1{+}...{+}n_6{*}w_6)
\end{array}
$$

Fig. 1.3: Exemple d'un système de contraintes linéaires issu de la méthode IPET. Les contraintes de flot modélisent la structure du graphe de flot de contrôle, le nombre d'itérations des boucles est borné par une constante (maxiter) et la fonction objectif à maximiser est la fréquence d'exécution de chacun des blocs de base multipliée par leur temps d'exécution pire cas.

À partir de ce système de contraintes, on cherche à maximiser l'expression suivante afin d'obtenir une borne supérieure du WCET :

$$\sum_i n_i * w_i$$

où w_i représente une borne supérieure du WCET du bloc de base i, fournie par l'analyse de bas niveau. La maximisation de cette expression est effectuée par un solveur de contraintes linéaires.

La figure 1.3 présente le système de contraintes linéaires obtenu pour le graphe de flot de contrôle de la figure 1.2.

Cette méthode, reposant uniquement sur le graphe de flot de contrôle obtenu à partir du code objet de l'application, est intéressante car elle permet de prendre en compte des codes optimisés lors de la phase de compilation. L'analyse de code optimisé peut tout de même s'avérer complexe car il faut être en mesure de détecter et fournir des informations supplémentaires comme le nombre maximum d'itération des boucles après optimisations. De plus, bien que la résolution d'un système

de contraintes linéaires en nombres entiers soit dans le cas général un problème NP-complet [93], la formulation de ce problème sous la forme de contraintes de flot permet de résoudre ce problème en temps polynomial [53, 47]. Finalement, la modélisation de ce problème sous forme de contraintes permet d'ajouter certaines propriétés comme par exemple celle des chemins infaisables [63] permettant ainsi de raffiner l'estimation du pire temps d'exécution.

1.4 Analyse bas niveau : analyse du comportement temporel pire cas des mémoires cache

Nous allons maintenant nous intéresser au travaux réalisés pour les mémoires cache dans un contexte d'utilisation temps-réel. Après une description de ce type de mécanisme, nous présentons les principales analyses existantes permettant de déterminer leur comportement temporel pire cas. Nous considérons, dans ce chapitre, uniquement les architectures constituées d'un *seul niveau* de cache, le cas des architectures disposant de plusieurs niveaux sera étudié dans la suite du document, celles-ci ne disposant que de peu de méthodes d'analyse du comportement temporel pire cas. Nous détaillons, en particulier, une méthode basée sur la théorie de l'interprétation abstraite, utilisée par la suite dans nos travaux présentés dans les chapitres 2 et 3. Nous finirons cette partie en regardant les effets sur le pire temps d'exécution résultant de certaines interactions avec d'autres mécanismes matériels. D'autres travaux, traitant de l'utilisation des mémoires cache dans un cadre temps-réel, seront abordés dans la section suivante.

1.4.1 Architectures des mémoires cache

Les mémoires cache ont été introduites pour réduire le temps d'accès aux informations dû à l'écart croissant entre le temps de calcul des micro-processeurs et le temps d'accès à la mémoire principale. Ce sont des mémoires à accès rapide de faible capacité, situées à proximité du processeur comparativement à la mémoire principale. Elles sont constituées de plusieurs blocs de taille fixe pouvant contenir une suite contiguë de mots mémoires appelée ligne de cache. Les caches sont très efficaces pour réduire le temps d'accès moyen aux informations en exploitant la localité spatiale et temporelle des applications. Lors de l'accès à une donnée, si celle-ci n'est pas présente dans une ligne de cache, on parle alors de *défaut de cache* et la ligne de cache contenant cette donnée est chargée à partir de la mémoire principale dans un bloc du cache. Les accès ultérieurs à cette même ligne de cache soit à la même donnée (localité temporelle) soit à une donnée présente au sein de la même ligne (localité spatiale) produisent un *succès de cache*, évitant ainsi un accès à la mémoire principale ce qui permet de réduire la latence d'accès mémoire.

Différentes structures de caches existent [96] : les caches à correspondance directe, les caches totalement associatifs et les caches associatifs par ensembles, comme illustré par la figure 1.4 (il s'agit d'une adaptation d'une illustration se trouvant dans [69]).

- pour les caches à correspondance directe, une ligne de cache donnée dispose d'un unique bloc de cache pouvant la stocker (figure 1.4.a) ;
- pour les caches totalement associatifs, une ligne de cache donnée peut être stockée dans n'importe quel bloc du cache (figure 1.4.b) ;
- pour les caches associatifs par ensembles, une ligne de cache donnée peut être stockée dans un nombre limité de blocs de cache. Par exemple, si le degré d'associativité est de deux alors une ligne de cache dispose de deux emplacements possibles dans le cache (figure 1.4.c).

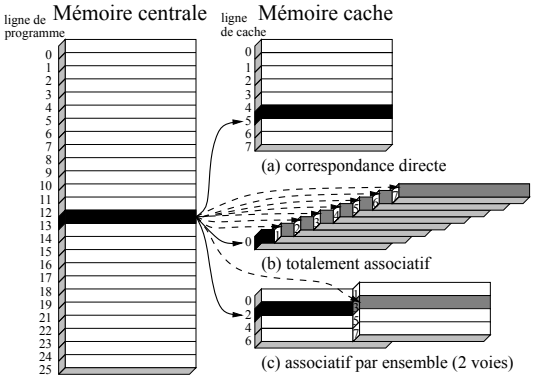


FIG. 1.4: Les différentes structures de mémoires cache où un mot mémoire peut être contenu dans un emplacement (a), dans tous les emplacements (b) ou dans un ensemble d'emplacements (c) du cache.

Les caches à correspondance directe sont similaires aux caches associatifs par ensembles dont le degré d'associativité est de un et les caches totalement associatifs sont similaires aux caches associatifs par ensembles dont le degré d'associativité est égal au nombre de bloc de cache. Cette vision permet de prendre en paramètre le degré d'associativité des caches et ainsi d'adapter plus facilement les différentes méthodes d'analyse.

Pour les caches associatifs, une classification des différents types de défauts de cache a été définie dans [44]. Elle est composée des trois catégories suivantes :

- Inévitable (en anglais *compulsory*) : ce type de défaut correspond au défaut provoqué lors du premier accès à une ligne de cache ;
- Capacité : ce type de défaut se produit lorsque la taille des données accédées par l'application dépasse celle du cache ;

– Conflit : ce type de défaut se produit lorsque de nombreuses lignes de cache sont projetées dans un même ensemble du cache et dépasse alors le degré d'associativité de ce dernier.

Pour ces types de cache, lors du chargement d'une ligne de cache, les autres lignes déjà présentes dans le même ensemble se trouvent en conflit dès lors que cet ensemble est plein. Il faut alors déterminer une ligne de cache à évincer. Pour ce faire, il existe différentes politiques de remplacement de cache dont :

– *Least Recently Used (LRU)* qui sélectionne la ligne de cache la moins récemment référencée pour être évincée. Cette politique est utilisée par exemple dans l'Intel Pentium I et le MIPS 24K/34K.
– *Pseudo-LRU* qui est basée sur le même principe que la politique LRU cependant, la notion d'âge est moins précise afin de réduire le nombre de bits utilisés par la politique de remplacement. L'âge est défini par un arbre binaire dont les nœuds ont pour valeur 0 ou 1 et les feuilles sont les blocs de cache d'un ensemble, comme illustré par la figure 1.5. La valeur 0 indique que le sous-arbre de droite est considéré comme le plus récent et la valeur 1 indique le contraire. Lors d'un accès, l'âge est mis à jour en inversant la valeur des nœuds traversés. Cette politique est utilisée par exemple dans le PowerPC 75x et l'Intel Pentium II-IV.
– *First-In First-Out (FIFO)* ou *Round-Robin* qui sélectionne la ligne de cache la plus anciennement insérée pour être évincée. Cette politique est utilisée par exemple dans l'Intel XScale, l'ARM9 et l'ARM11.
– *Most Recently Used (MRU)* comme définie dans [3], utilise un bit pour chaque ligne de cache. A chaque fois qu'une ligne est accédée, la valeur de son bit est mise à 1, indiquant que la ligne a été récemment accédée. Lorsque le dernier bit de l'ensemble est mis à 1, tous les autres bits sont remis à 0. Lors d'un défaut de cache, la ligne de cache ayant son bit à 0 et disposant de l'index le plus bas (nous considérerons celui le plus à gauche par la suite) est évincée.
– *Random* qui sélectionne de façon aléatoire la ligne de cache à évincer.

La présentation de ces différentes politiques de remplacement de cache n'a pas pour vocation d'être exhaustive. Elle se limite aux politiques que nous étudierons et analyserons dans ce document. Bien d'autres politiques de remplacement de caches ont été proposées comme par exemple *Dynamic Insertion Policy (DIP)* [82]. La recherche de politique de remplacement de cache ainsi que de structures de caches [95] améliorant les performances est toujours un domaine actif [19] sachant que la seule politique de remplacement montrée optimale [12] est impossible à mettre en œuvre car elle nécessite une connaissance précise des réutilisations futures.

Les mémoires cache peuvent contenir le code du programme (cache d'instructions), les données (cache de données) ou les deux (cache unifié). Une architecture

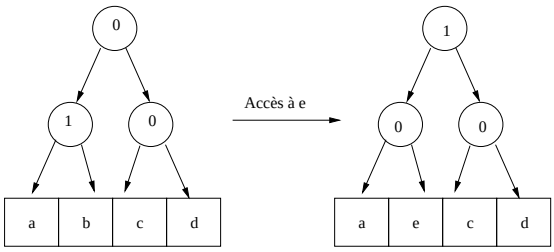


FIG. 1.5: La notion d'âge est représentée sous la forme d'un arbre binaire pour la politique de remplacement de cache PLRU. Lors d'un accès, l'âge est mis à jour en inversant la valeur de chacun des nœuds traversés.

disposant de caches séparés signifie que le processeur dispose d'un cache d'instructions et d'un cache de données distincts. Dans cette étude, nous étudierons principalement les caches d'instructions. Néanmoins, l'application des méthodes proposées aux caches de données sera abordée succinctement à la fin de chaque chapitre.

Les caches de données et les caches unifiés diffèrent des caches d'instructions en raison des accès en écriture aux données lors de l'exécution. Un mécanisme supplémentaire est utilisé pour traiter les écritures, mettant en œuvre l'une des deux politiques suivantes :

- Propagation de l'écriture (en anglais : *write through*) : lors d'une écriture, à la fois le cache et la mémoire principale sont mis à jour ;
- Écriture différée (en anglais : *write back*) : lors d'une écriture, seulement le cache est mise à jour, la mémoire principale est mis à jour lors de l'éviction de la ligne de cache modifiée.

Pour traiter le cas où la ligne de cache contenant l'information à modifier est absente du cache, deux comportements ont été définis :

- Écriture sans allocation (en anglais : *write-no-allocate*) : l'information modifiée n'est pas insérée dans le cache ;
- Écriture avec allocation (en anglais : *write-allocate*) : l'information modifiée est insérée dans le cache.

En résumé, les mémoires cache permettent d'accélérer le temps moyen d'exécution des applications en conservant les informations récemment accédées afin de réduire le temps d'accès aux informations lors de leur réutilisation en exploitant la localité spatiale et temporelle des applications. Différentes architectures ainsi que différentes politiques de remplacement de cache ont été proposées. Nous avons présenté les plus courantes que nous étudierons dans ce document, en nous intéressant plus particulièrement à l'analyse de leur comportement temporel pire cas lors de l'estimation du pire temps d'exécution.

1.4.2 Mémoires cache et estimation du pire temps d'exécution

La difficulté principale lors de l'estimation du pire temps d'exécution est de déterminer à priori le comportement temporel pire cas du cache pour chaque accès mémoire. En effet, le contenu des caches est géré de façon dynamique et fortement corrélé au chemin emprunté lors de l'exécution. De ce fait, déterminer si un accès produira un succès ou un défaut de cache lors de l'exécution et ce de façon précise peut s'avérer complexe. L'analyse étant réalisée hors-ligne, elle doit considérer l'ensemble des chemins d'exécution possibles pour être indépendante des données d'entrées et ainsi garantir la sûreté de l'estimation. Une solution simple pour traiter cette problématique serait de considérer un défaut de cache pour chaque accès mémoire[1] mais de cette façon, l'estimation du pire temps d'exécution serait inexploitable car elle sur-approximerait de façon trop importante le pire temps d'exécution réel. De nombreux travaux ont été réalisés durant les quinze dernières années afin de prendre en compte de façon précise l'impact des caches lors de l'estimation du pire temps d'exécution. Nous verrons dans un premier temps les méthodes d'analyse statique puis nous regarderons ensuite différentes approches traitant des mémoires cache lors de l'estimation du pire temps d'exécution.

1.4.3 Tour d'horizon des méthodes d'analyse statique du comportement temporel pire cas des mémoires cache

Pour garantir la sûreté de l'estimation du pire temps d'exécution, les méthodes d'analyse statique déterminent l'ensemble de tous les contenus de cache possibles en chaque point de l'application en considérant tous les chemins d'exécution. La représentation de l'ensemble de tous les contenus possibles est réalisée soit par un ensemble d'états dits *concrets de cache* [71] soit par une représentation plus compacte appelée *état abstrait de cache* [67, 69, 34, 101] (en anglais : *Abstract Cache State (ACS)*). La représentation sous la forme d'états concrets de cache est la plus précise car elle permet d'énumérer l'ensemble de tous les contenus de cache possibles issus de chacun des chemins d'exécution. Cependant, cette énumération est très coûteuse en temps et en consommation mémoire, en raison du nombre exponentiel de chemins possibles même si le nombre d'états peut être inférieur au nombre de chemins. Les états abstraits de cache utilisent, quand à eux, une vision ensembliste des contenus possibles pour représenter dans une même structure de donnée le contenu de l'ensemble des chemins d'exécution en perdant de fait la notion d'exhaustivité. Concrètement, un état abstrait de cache peut être vu comme une union de tous les états concrets de cache possibles.

[1] Un défaut de cache représente le temps d'accès pire cas pour les architectures sans anomalies temporelles (cf section 1.4.6).

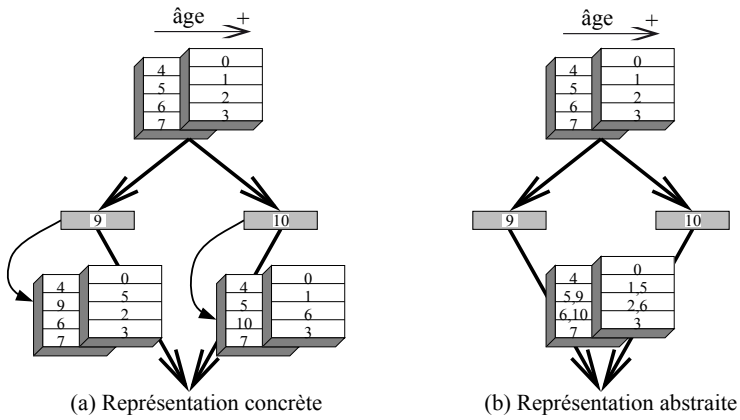


FIG. 1.6: Représentation concrète et abstraite des caches.

La figure 1.6 (il s'agit d'une adaptation d'une illustration se trouvant dans [69]) illustre la différence entre les représentations concrète et abstraite d'un cache associatif par ensembles de deux voies utilisant une politique de remplacement LRU. Sur le chemin d'exécution de gauche, la référence mémoire 9 est chargée dans le cache tandis que la référence 10 est chargée sur le chemin de droite amenant à deux états possibles du cache (1.6.a). La représentation abstraite permet de regrouper ces deux états concrets en un seul état abstrait (1.6.b) en réalisant l'union des deux états sur chaque ensemble et chaque voie du cache.

Les premiers travaux adressant l'impact des mémoires cache sur l'estimation du pire temps d'exécution se sont focalisés sur l'analyse du contenu des caches d'instructions. Principalement deux méthodes d'analyse statique ont été définies [67, 34], basées sur un calcul de point fixe.

La première méthode à avoir été proposée est communément appelée *simulation statique de cache* [67]. Elle calcule des états abstraits de cache en utilisant une analyse de flot de données permettant de déduire le contenu abstrait du cache en chaque point de programme et ainsi déterminer si un accès produira un succès ou un défaut de cache lors de l'exécution. Cette méthode, initialement définie pour les caches à correspondance directe, a été par la suite étendue aux caches associatifs par ensembles disposant d'une politique de remplacement de cache LRU [69].

La seconde méthode [34] repose sur la théorie de l'interprétation abstraite [25, 26]. Nous expliquerons en détail le fonctionnement de cette méthode dans la section 1.4.4, car nos travaux présentés dans les chapitres 2 et 3 de ce document sont basés sur cette méthode, principalement pour les théorèmes de l'interprétation

abstraite sur lesquels elle repose. Cette méthode, initialement définie pour des caches associatifs par ensembles disposant d'une politique de remplacement de cache LRU, a été étendue par la suite à différentes politiques de remplacement de cache [43]. Cette méthode a également été améliorée dans [9] en analysant chacun des niveaux de boucle afin de raffiner l'estimation du pire temps d'exécution.

Ces méthodes d'analyse statique reposant sur des calculs de point fixe peuvent s'avérer coûteuses en terme de temps de calcul pour des applications de taille importante. Une solution pour améliorer le passage à l'échelle de ces méthodes est de réaliser une analyse statique de cache partielle [10, 8]. L'idée de cette approche consiste à analyser le programme par partie, puis à combiner les résultats afin d'obtenir le comportement pire cas des accès au cache. Cette méthode permet de réduire de façon significative le temps de calcul de l'analyse en ne révélant qu'une légère perte de précision lors de l'estimation du pire temps d'exécution.

Ces travaux ont également été étendus pour analyser le contenu des caches de données dans [105] pour la méthode de simulation statique de cache et dans [35, 94] pour la méthode reposant sur l'interprétation abstraite.

La difficulté supplémentaire pour analyser le contenu des caches de données provient du fait qu'un accès mémoire peut référencer une plage d'adresses, comme par exemple le parcours d'un tableau au sein d'une boucle. Il faut donc d'une part déterminer statiquement les adresses référencées (ou un sur-ensemble). Pour ce faire, des analyses peuvent être effectuées sur le code assembleur [105, 39] ou directement ajoutées dans le compilateur [29, 30]. D'autre part, pour obtenir une analyse précise, il faut tenir compte des schémas d'accès aux données (accès réguliers, accès dépendants des données d'entrées...).

Pour analyser finement le comportement des caches de données pour ces types d'accès, une méthode proposant des équations de défauts de cache (en anglais : *Cache Miss Equations*) a été explorée dans [36, 83]. L'approche consiste à représenter l'espace d'itération des boucles sous la forme de polyèdres et à utiliser des vecteurs de réutilisation entre les différents points de cet espace modélisés ensuite par des équations. En résolvant ces équations, il est possible de déterminer précisément les succès et les défauts de cache se produisant lors de l'exécution.

1.4.4 Méthode d'analyse statique pour les caches d'instructions basée sur la théorie de l'interprétation abstraite

Cette méthode d'analyse statique, proposée dans [34, 101], consiste à classifier chaque référence mémoire par son comportement pire cas dans le cache lors de l'exécution. Pour ce faire, une classification que nous appellerons *classification de comportement* est définie de la façon suivante :

- *Always-Hit (AH)* : la référence produira toujours un succès de cache lors de l'exécution ;
- *Always-Miss (AM)* : la référence produira toujours un défaut de cache lors de l'exécution ;
- *First-Miss (FM)* : la référence pourra produire un succès ou un défaut de cache lors du premier accès tandis que les accès suivants produiront toujours un succès de cache lors de l'exécution ;
- *Not-Classified (NC)* : tous les autres cas.

Afin de déterminer la classification de comportement de chaque accès à une référence mémoire, l'analyse de cache définit trois analyses point-fixe, manipulant des états abstraits de cache, appliquées sur le graphe de flot de contrôle de l'application.

- L'analyse *Must* détermine si une référence mémoire est toujours contenue dans le cache à un point de programme donné ;
- L'analyse *Persistence* détermine si une référence mémoire ne sera pas évincée après avoir été préalablement chargée dans le cache à un point de programme donné ;
- L'analyse *May* détermine si une référence mémoire peut être contenue dans le cache à un point de programme donné.

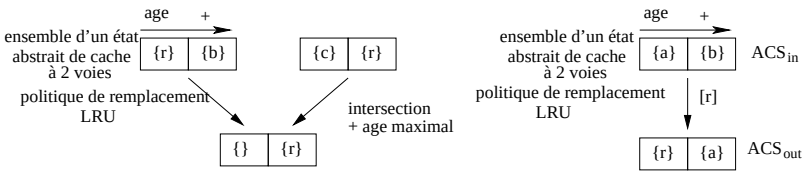
Chacune de ces trois analyses calcule des états abstraits de cache pour chacun des blocs de base. Pour ce faire, deux fonctions dans le domaine abstrait nommées *Update* et *Join* sont définies pour chacune des analyses :

- La fonction *Update* est appelée lors de chaque accès à une référence mémoire et considère l'état abstrait de cache avant l'accès (ACS_{in}) et calcule l'état abstrait de cache résultant de l'accès à cette référence mémoire (ACS_{out}) en respectant la politique de remplacement de cache ainsi que la sémantique de l'analyse ;
- La fonction *Join* est utilisée pour fusionner deux états abstraits de cache lorsqu'un bloc de base dispose de deux prédécesseurs dans le graphe de flot de contrôle comme par exemple au point de re-convergence d'une conditionnelle.

La figure 1.7 présente un exemple d'utilisation de la fonction *Join* (1.7.a) et *Update* (1.7.b) pour l'analyse *Must* d'un cache associatif par ensembles disposant d'un degré d'associativité de deux et utilisant la politique de remplacement LRU. Comme cette politique est indépendante pour chacun des ensembles du cache, seulement l'ensemble concerné est illustré. Une notion d'âge est associée à chacun des blocs de cache de cet ensemble. Plus l'âge d'un bloc est petit, plus la ligne de cache contenue dans ce bloc a été accédée récemment. Pour l'analyse *Must*, une référence mémoire r est stockée une seule fois dans l'état abstrait de cache avec son âge maximal. Ceci signifie que lors de l'exécution la référence mémoire r aura un âge toujours inférieur ou égal à l'âge déterminé lors de l'analyse *Must*. Les

fonctions *Join* et *Update* sont définies de la façon suivante pour l'analyse *Must* :

- La fonction *Join* est appliquée sur deux états abstraits de cache et retourne un état abstrait de cache contenant uniquement les références mémoire présentes dans les deux états abstraits de cache d'entrée, en conservant leur âge maximal ;
- La fonction *Update* effectue l'accès à une référence mémoire r sur un état abstrait de cache et retourne l'état de cache résultant de cet accès. Cette fonction projette la référence r dans l'ensemble du cache correspondant et lui associe l'âge le plus petit et augmente l'âge des références mémoire présentes dans l'état abstrait de cache d'entrée. Lorsque l'âge d'une référence mémoire devient supérieur au degré d'associativité du cache, la référence mémoire est évincée de l'état abstrait de cache.



a. Fonction Join de l'analyse Must. b. Fonction Update de l'analyse Must.

FIG. 1.7: Exemple d'utilisation des fonctions *Join* et *Update* pour l'analyse *Must* avec la politique de remplacement de cache LRU. La fonction *Join* conserve l'âge maximal des références mémoire présentes dans les deux états abstraits de cache tandis que la fonction *Update* effectue la mise à jour du contenu du cache lors d'un accès à la référence mémoire r.

Pour les analyses *May* et *Persistence*, l'approche est similaire et la fonction *Join* est définie de la façon suivante :

- Analyse *May* : la fonction *Join* réalise l'union des références mémoire présentes dans les états abstraits de cache et conserve l'âge minimal ;
- Analyse *Persistence* : la fonction *Join* réalise l'union des références mémoire présentes dans les états abstraits de cache et conserve l'âge maximal. Pour ce faire une voie virtuelle est ajoutée à chacun des ensembles du cache pour conserver l'ensemble des références mémoire potentiellement évincées.

Finalement, la classification de comportement de l'accès à une référence mémoire se déduit en fonction du contenu des états abstraits de cache de chacune des analyses avant l'accès à cette référence. Si la référence mémoire est contenue dans l'état abstrait de cache de l'analyse *Must*, la classification de cette référence est *Always-Hit*. Sinon, si elle est présente dans l'état abstrait de l'analyse *Persistence*, la classification est *First-Miss*. Sinon, si elle est présente dans l'état abstrait de l'analyse *May*, la classification est *Not Classified* tandis que si elle en est absente, la classification est *Always-Miss*.

1.4.5 Résultats théoriques permettant la prise en compte de différentes politiques de remplacement de cache

Concernant l'estimation du pire temps d'exécution pour des architectures disposant de mémoires cache, des études théoriques sur le comportement pire cas de ces mémoires ont également été menées en s'intéressant principalement aux différentes politiques de remplacement de cache.

Pour certaines politiques de remplacement de cache tel que Pseudo-LRU, [13] montre que partir de l'hypothèse d'un cache vide avant l'exécution de l'application ne conduit pas nécessairement au comportement pire cas, en raison d'un phénomène d'effet domino possible entre les itérations successives des boucles. L'hypothèse d'un cache vide est néanmoins exploitable si on considère que le contenu du cache est intégralement vidé avant l'exécution de la tâche, au prix d'un surcoût en temps supplémentaire, sachant que cette fonctionnalité est présente dans la plupart des processeurs actuels.

D'autres études se sont intéressées à la prévisibilité des différentes politiques de remplacement de cache [86]. Ils formalisent le résultat intuitif que la politique de remplacement de cache LRU est la plus prévisible dans le cadre des analyses de comportement pire cas en se basant la durée de vie minimale et maximale d'une ligne de cache lors d'une séquence d'accès pour les politiques de remplacement de cache LRU, pseudo-LRU, FIFO et MRU. Dans [85] des ratios de compétitivité sont définis entre ces politiques afin de déterminer le nombre de défauts de cache d'une politique de remplacement à partir des résultats de l'analyse de cache de la politique LRU. Cette solution permet de déterminer le nombre de défauts de cache dans le pire cas sans pour autant fournir d'informations sur l'endroit dans le code où ils se produisent.

1.4.6 Interaction entre mémoire cache et *pipeline*

La classification de comportement de chacune des instructions déterminée par l'analyse permet d'évaluer leur temps d'accès. Dans le cas d'une classification représentant un succès de cache (i.e. classification de comportement *Always-Hit*), le temps d'accès correspond à la latence du cache tandis que si la classification représente un défaut de cache (i.e. classification de comportement *Always-Miss*), le temps d'accès correspond à la latence du cache plus la latence de la mémoire principale. Intuitivement, pour l'estimation du pire temps d'exécution, il semble donc naturel de considérer la latence correspondant à un défaut de cache lorsque l'accès peut produire un succès ou un défaut de cache (i.e. classification de comportement *Not Classified*).

Cependant, cette hypothèse a été détectée dans [61] comme pouvant mettre en défaut la sûreté de l'estimation du pire temps d'exécution, suivant l'architecture

sur laquelle la tâche est exécutée. En effet dans certains cas, le pire temps local d'une instruction ne mène pas nécessairement au pire temps global de la séquence d'instruction la contenant. Pour illustrer ce phénomène, prenons une architecture disposant d'un *pipeline* à exécution dans le désordre et d'une mémoire cache. La tâche considérée est constituée de quatre instructions notée i_1, i_2, i_3 et i_4. Les instructions i_1 et i_4 sont des accès mémoire et utilisent l'unité fonctionnelle du processeur notée uf_1 tandis que les instructions i_2 et i_3 s'exécute sur l'unité fonctionnelle uf_2. i_1 est classifiée *Not Classified* pouvant donc produire un succès ou un défaut de cache lors de l'exécution. Finalement, il y a une dépendance de données entre i_1 et i_2 ainsi qu'entre i_3 et i_4. La figure 1.8 illustre l'exécution de cette séquence d'instructions lors d'un défaut de cache pour l'instruction i_1 (figure 1.8.a) et lors d'un succès de cache (figure 1.8.b).

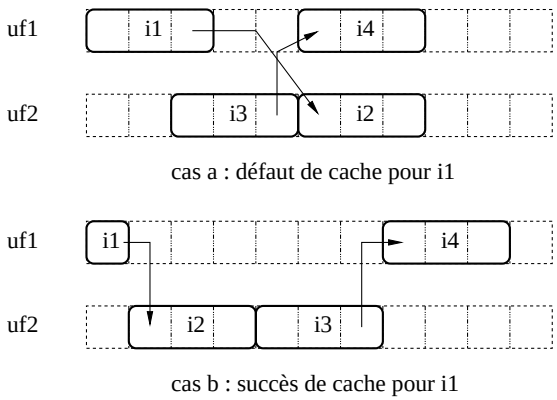


FIG. 1.8: Exemple d'anomalie temporelle résultant de l'interaction entre un cache et un pipeline. Lorsque l'accès à l'instruction $i1$ produit un défaut de cache, le temps d'exécution de la séquence d'instructions est globalement plus court que lorsque l'accès produit un succès de cache en raison de l'ordonnancement des instructions.

Dans le cas où l'instruction i_1 produit un défaut de cache, i_3 est exécutée avant i_2 qui attend la fin de i_1 (dépendance de donnée) permettant ainsi d'exécuter i_4 et i_2 simultanément. Lorsque i_1 produit un succès par contre, l'instruction i_2 peut être exécutée directement ce qui retarde l'exécution de i_3 comparativement au cas précédent et produit donc un temps d'exécution plus grand pour cette séquence d'instructions. Sur cet exemple, nous avons donc bien un cas où un défaut de cache ne conduit pas au pire temps d'exécution.

Différents facteurs peuvent être à l'origine d'anomalies temporelles dont l'ordonnancement des instructions, les mécanismes de pré-chargement et certaines politiques de remplacement de cache [87]. Dans ce document, nous considérons que les tâches sont exécutées sur des architectures non sujettes aux anomalies temporelles [104].

Des travaux ont néanmoins été proposées pour prendre en compte ce type de comportement lors de l'estimation du pire temps d'exécution [32, 51] ou pour éviter qu'ils se produisent [89, 77].

1.5 Autres travaux relatifs aux caches pour les systèmes temps-réel

Dans cette partie, nous allons présenter différentes méthodes relatives aux caches et à leur utilisation dans le cadre des systèmes temps-réel. Nous verrons d'autres méthodes orientées analyse statique puis nous regarderons des solutions basées sur des mécanismes matériels ainsi que des approches orientées compilation afin d'améliorer la prévisibilité du comportement des mémoires cache.

1.5.1 Solutions à base d'analyse statique

Alternatives à l'analyse découplée bas niveau/haut niveau

L'estimation du pire temps d'exécution est généralement réalisée par une analyse bas niveau découplée de l'analyse haut niveau. Il existe cependant des méthodes d'analyse regroupant ces deux phases en une seule.

L'analyse proposée dans [54] augmente l'approche d'énumération implicite des chemins en ajoutant des contraintes linéaires modélisant le comportement d'un cache à correspondance directe en chaque point de programme. Les défauts de cache et les succès de cache sont modélisés par des variables binaires associées à chaque accès mémoire et sont contraints par une équation représentant le fait que deux références mémoire projetées dans le même emplacement du cache ne peuvent être présentes en même temps dans le cache en un point de programme. Ainsi, cette approche permet d'estimer le pire temps d'exécution d'une tâche en une seule étape consistant à résoudre le système de contraintes linéaires. Cette méthode, bien que très précise, n'a pas été étendue aux caches associatifs par ensembles en raison du temps de calcul très important qu'elle nécessite même pour des programmes de taille réduite.

Une autre approche, appelée communément *exécution symbolique* [59, 15], consiste à exécuter le programme sur un simulateur modifié pour prendre en compte l'ensemble des chemins et ce indépendamment des données d'entrées. La sémantique de chacune des instructions est conservée par ce type d'approche permettant ainsi d'éviter les chemins infaisables, l'annotation du nombre d'itérations maximum des boucles étant dérivée directement par l'exécution symbolique et rendant de ce fait l'estimation du pire temps d'exécution précise. Néanmoins comme l'approche précédente, cette méthode est très coûteuse en temps de calcul.

Analyse des délais de préemption liés au mémoire cache

La prise en compte des caches dans l'analyse des systèmes temps-réel multi-tâches préemptifs ajoute une certaine complexité en raison des interférences inter-tâches. Pour illustrer ce propos, considérons la préemption d'une tâche t_1 par une tâche plus prioritaire t_2. Les contenus de cache lors de la préemption de $t1$ et à la fin de l'exécution de t_2 peuvent être différents, car les lignes de cache utilisées par la tâche t_2 peuvent évincer celles utilisées par t_1. Lors de la reprise de la tâche t_1, des défauts de cache supplémentaires sont causés par ces interférences. Le temps supplémentaire introduit par ces défauts de cache est couramment appelé *Cache-Related Preemption Delay (CRPD)*.

Plusieurs travaux [71, 98, 97, 84, 16] se sont intéressés à l'estimation d'une borne supérieure du CRPD. Globalement, ils proposent une analyse statique reposant d'une part sur la connaissance du contenu possible du cache en chaque point de programme de la tâche préemptée, et d'autre part sur les lignes de cache utilisées par la tâche plus prioritaire. Le CRPD de la tâche préemptée, en un point de programme, est ensuite calculé en considérant le nombre de lignes de cache potentiellement réutilisées après la préemption et potentiellement évincées lors de l'exécution de la tâche plus prioritaire, multiplié par le coût d'un défaut de cache. Le CRPD d'une tâche se dérive ensuite en prenant la valeur maximale du temps de préemption entre tous les points du programme. [84] raffine cette estimation en prenant en compte le nombre de fois où la tâche peut être préemptée pendant son exécution. Pour sa part, [98] intègre le fait qu'une préemption peut être causée par plusieurs tâches simultanément afin d'affiner l'estimation.

1.5.2 Approches matérielles pour plus de déterminisme

Partitionnement de cache

La méthode de partitionnement de cache proposée dans [92] consiste à diviser le cache en partitions de taille fixe puis, à assigner un nombre fixe de partitions à chacune des tâches s'exécutant sur le système. Chaque tâche dispose ainsi d'un espace réservé à elle seule dans le cache, évitant ainsi les conflits inter-tâches dans les systèmes multi-tâches préemptifs. Cette méthode, supprimant ce type de conflits, permet de rendre les délais de préemptions (CRPD) nuls. Cependant, cette approche a un impact sur l'utilisation du cache et sur le temps d'exécution car chaque tâche dispose alors d'un volume de cache plus limité. Une analyse de cache reste nécessaire pour estimer le comportement temporel pire cas des accès mémoire au sein de la partition assignée à une tâche.

Gel du contenu du cache

La méthode de gel du contenu du cache consiste à charger le cache avec des références mémoire définies lors de la phase de compilation puis, à le figer pour qu'il reste inchangé sur une zone du programme donnée lors de l'exécution. L'idée principale de cette approche consiste à rendre prévisible le temps d'accès mémoire en disposant statiquement du contenu du cache en chaque point de programme.

Dans les travaux existants, deux approches ont été proposées : le gel de cache statique (en anglais : *static cache locking*) [20, 78] et le gel de cache dynamique (en anglais : *dynamic cache locking*) [102, 77]. Le gel de cache statique consiste à charger le contenu du cache à l'initialisation du système. Celui-ci reste ensuite figé durant toute l'exécution. Le gel de cache dynamique quant à lui, consiste également à charger le contenu du cache au démarrage mais, le contenu du cache est en plus modifié lors de l'exécution par exemple entre deux régions de code contenant des nids de boucles. Dans les deux cas, le contenu du cache est déterminé de façon hors-ligne et vise à réduire l'estimation du pire temps d'exécution en déterminant le contenu à charger en fonction de son utilisation sur le pire chemin d'exécution.

Le concept de gel du contenu du cache permet de connaître statiquement le contenu du cache en chaque point de programme. La connaissance de ce contenu permet de prédire de façon sûre et précise le temps d'accès aux données et les délais de préemptions. Cela permet également de supprimer les anomalies temporelles identifiées précédemment (paragraphe 1.4.6). Enfin, l'utilisation de cette méthode permet d'améliorer l'estimation du pire temps d'exécution comparativement à l'estimation issue d'une analyse du comportement pire cas dans le cas de caches utilisant une politique de remplacement pseudo-aléatoire ou non documentée disposant des fonctionnalités permettant le chargement et le gel de cache.

1.5.3 Compilation orientée pire temps d'exécution

Différentes optimisations de compilation ont été proposées afin d'améliorer l'estimation du pire temps d'exécution. Concernant les mémoires cache, les optimisations proposées visent à améliorer leur utilisation en essayant par exemple de réduire les défauts de cache liés aux conflits. Pour ce faire [56] propose une méthode de placement des procédures en se basant sur le graphe d'appel du programme afin de limiter les conflits sur le pire chemin d'exécution. D'autres optimisations ont également été proposées comme la duplication de code [57], de chemin [111], le déroulage de boucle [58, 111] afin d'améliorer l'estimation du pire temps d'exécution.

Une autre approche proposée dans [60] pour les caches de données consiste à ne stocker qu'une partie des références accédées dans le cache. Lors de l'analyse

de cache, une instruction référençant une donnée dont l'adresse est calculée à l'exécution, comme par exemple l'accès à un tableau, est considérée comme un accès indéterministe et n'a pour effet que de polluer le contenu du cache. L'idée présentée dans [60] consiste à ne pas stocker ce type de donnée afin d'éviter cette pollution et ainsi réduire les conflits et donc l'estimation du pire temps d'exécution.

Finalement, une approche visant également à réduire les défauts de cache a été proposée dans [72]. L'idée de cette méthode ne consiste pas à réduire les conflits mais à réduire le nombre d'accès à la mémoire en compressant le code de l'application. La décompression est réalisée entre le chargement et le décodage de l'instruction, ce qui permet de stocker le code compressé au niveau du cache et ainsi réduire le nombre d'accès à la mémoire. Bien que la décompression des instructions prenne du temps, la réduction des accès mémoire est telle que l'utilisation de cette technique permet dans la plupart des cas de réduire l'estimation du pire temps d'exécution.

1.6 Discussion

Dans ce chapitre, nous venons de détailler les principales méthodes d'estimation de pire temps d'exécution pour des applications temps-réel exécutées sur des architectures disposant de mémoires cache. Après avoir défini les propriétés requises par l'estimation du pire temps d'exécution à savoir la sûreté et la précision, nous avons décrit les deux classes de méthodes d'estimation, en développant deux méthodes statiques, l'une à base d'arbres syntaxiques et l'autre utilisant une formulation sous la forme de contraintes linéaires en nombres entiers.

Nous sommes ensuite rentrés un peu plus dans le sujet de ce document en regardant les travaux existants pour les mémoires cache. Après une présentation de ce mécanisme matériel et de ces diverses mises en œuvre, nous avons détaillé une méthode d'analyse statique du contenu des caches d'instructions basée sur la théorie de l'interprétation abstraite. Nous avons également évoqué les interactions possibles et les effets contre-intuitifs vis-à-vis du comportement pire cas des caches lorsqu'ils sont utilisés simultanément avec d'autres mécanismes matériels comme les pipelines. Différentes approches visant à améliorer l'estimation du pire temps d'exécution ont ensuite été abordées comme par exemple le gel de contenu de cache, le partitionnement des caches.

L'ensemble de ces méthodes sont définies dans la littérature pour des architectures matérielles disposant d'un seul niveau de mémoire cache tandis que les architectures contemporaines disposent généralement d'une hiérarchie de mémoires cache. Peu de travaux ce sont intéressés au cas des hiérarchies de caches.

Nous verrons dans le chapitre 2 que l'unique méthode existante visant à ana-

lyser des hiérarchies de mémoires cache [68] peut mettre en défaut la propriété de sûreté nécessaire à la validation des systèmes temps-réel. En partant de cette constatation, nous proposons dans ce même chapitre une méthode d'analyse statique du contenu des hiérarchies de caches d'instructions en considérant différents modes de gestion ainsi que différentes politiques de remplacement de cache.

Une autre problématique provenant de l'utilisation des processeurs multi-cœurs dans le cadre des systèmes temps-réel est celle des ressources partagées, dont notamment les mémoires cache. La présence de niveaux de caches partagés entre différents cœurs introduit une nouvelle source d'indéterminisme lors de l'estimation du pire temps d'exécution, provoquée par l'utilisation concurrente de ces ressources. Là encore, nous verrons dans le chapitre 3 que la seule méthode d'analyse statique existante au démarrage de cette thèse peut mettre en défaut la sûreté de l'estimation du pire temps d'exécution.

En se basant sur l'analyse proposée dans le chapitre 2, nous proposerons dans le chapitre 3 une méthode d'analyse statique sûre pour les niveaux de cache partagés, cumulée à une technique d'optimisation réalisée lors de la phase de compilation pour limiter les conflits dans ces niveaux de caches.

Chapitre 2

Analyse du contenu des hiérarchies de caches

2.1 Introduction

Les mémoires cache ont été introduites pour réduire le temps d'accès aux informations en raison de l'écart croissant entre la fréquences des microprocesseurs et la latence d'accès à la mémoire principale. Les caches sont très efficaces pour réduire le temps d'accès moyen aux informations en exploitant les propriétés de localité spatiale et temporelle des applications. Néanmoins, l'augmentation de la taille des mémoires cache a pour effet d'augmenter également la latence d'accès aux informations [96]. Pour pallier cette limite, les hiérarchies de mémoires cache ont été introduites [100]. Elles disposent d'un premier niveau de cache, proche du processeur, de petite capacité avec une faible latence d'accès, permettant ainsi de répondre rapidement aux requêtes mémoire du processeur. Ce niveau est suivi d'un ou plusieurs niveaux de caches, que nous appelons niveaux inférieurs, disposant d'une plus grande capacité mais d'une latence d'accès plus importante. Ces niveaux permettent de réduire les latences d'accès aux données moins fréquemment utilisées, comparativement à un accès systématique à la mémoire principale.

Afin de gérer le contenu des différents niveaux de caches composant de telles hiérarchies, différentes politiques de gestion de la hiérarchie mémoire ont été proposées :

- Politique de gestion *non-inclusive*. Lors d'un défaut de cache, dans les hiérarchies de caches non-inclusives, la ligne de cache contenant l'information demandée est systématiquement chargée dans tous les niveaux où le défaut se produit. Dans ce type de hiérarchie, bien qu'il n'y ait pas de mécanisme pour forcer l'inclusion, la plupart des informations contenues dans le premier niveau sont également contenues dans les niveaux de caches inférieurs.
- Politique de gestion *inclusive*. Dans les hiérarchies de caches inclusives, l'en-

semble des informations contenues dans un niveau de cache est également contenu dans chacun des niveaux de caches inférieurs. Lors d'un défaut de cache, le comportement est similaire à celui des hiérarchies de caches non-inclusives, la ligne de cache contenant l'information demandée est chargée dans tous les niveaux où le défaut se produit. Par contre, en cas d'éviction d'une ligne de cache dans un niveau de cache, la ligne est invalidée dans tous les niveaux supérieurs pour forcer l'inclusion. L'intérêt de ce type de hiérarchie est de simplifier le maintien de la cohérence des données. Par exemple dans un système multiprocesseurs, lorsqu'une ligne de cache doit être supprimée, il suffit de l'invalider dans le dernier niveau de cache, les autres niveaux étant traités par le mécanisme forçant l'inclusion.

– Politique de gestion *exclusive*. Dans les hiérarchies de caches exclusives, l'ensemble des informations contenues dans la hiérarchie est stocké dans *un seul niveau* de cache. Cette gestion permet d'éviter la duplication d'information dans les différents niveaux de caches et ainsi d'augmenter virtuellement la capacité de stockage de la hiérarchie. De ce fait, les hiérarchies de caches exclusives peuvent stocker plus d'informations que les hiérarchies inclusives et non-inclusives.

Ces différentes politiques de gestion sont couramment utilisées dans les processeurs généralistes. Citons par exemple les processeurs Intel Pentium II, III, IV qui mettent en œuvre une hiérarchie de caches non-inclusive ou les processeurs AMD Athlon qui mettent en œuvre une hiérarchie de caches exclusive. D'autres processeurs encore utilisent différentes politiques de gestion le long de la hiérarchie. Ainsi, le processeur IBM Power5 utilise une politique de gestion inclusive entre les deux premiers niveaux tandis que le dernier niveau dispose d'une politique de gestion exclusive avec les deux premiers niveaux.

Concernant les systèmes temps-réel embarqués, les architectures équipées de mémoires cache sont maintenant couramment utilisées en raison des besoins grandissants en terme de puissance de calcul. Comme nous l'avons vu précédemment, de nombreux travaux ont été réalisés pour estimer le pire temps d'exécution des applications exécutées sur des architectures équipées de mémoires cache (cf chapitre 1) mais en se limitant à l'analyse d'une architecture composée d'un unique niveau de cache. L'unique approche visant à analyser des hiérarchies de mémoires cache a été proposée dans [68] pour des hiérarchies de caches d'instructions non-inclusives utilisant la politique de remplacement LRU. Nous verrons dans la suite de ce chapitre que cette analyse pose des problèmes de sûreté lors de l'analyse de caches associatifs par ensembles.

2.1.1 Contributions

Dans ce chapitre, nous proposons une méthode d'analyse statique pour les hiérarchies de caches d'instructions. Nous définissons une analyse pour chacune des politiques de gestion [40, 42]. Une extension de ces analyses aux hiérarchies de caches de données [49] sera également étudiée à la fin de ce chapitre. Chacune des analyses est définie en considérant une politique de remplacement de cache LRU pour chacun des niveaux constituant la hiérarchie, nous verrons cependant que ces analyses sont à même de prendre en compte les politiques de remplacement de cache non-LRU les plus courantes.

Nous verrons que la prise en compte de l'ensemble des niveaux composant les hiérarchies de caches nous permet d'obtenir une estimation du pire temps d'exécution plus précise comparativement au fait de considérer un défaut de cache pour chacun des accès propagés dans les niveaux inférieurs. Ce type d'analyse nous servira également de base à l'analyse des processeurs multi-cœurs disposant généralement d'une hiérarchie de caches dont un sous-ensemble est partagé entre les différents cœurs.

2.1.2 Hypothèses et notations

Nous considérons une hiérarchie constituée de $nbLevels$ niveaux de caches d'instructions. Le niveau le plus haut dans la hiérarchie représente le cache interne (cache L1) qui est le plus proche du processeur. Les niveaux de caches sont numérotés de 1 (le plus haut dans la hiérarchie) à $nbLevels$ (le plus bas dans la hiérarchie). Concernant leur structure, chaque cache est associatif par ensembles. Cette structure est la plus générale et permet de modéliser également les caches totalement associatifs avec un degré d'associativité égal au nombre de lignes de cache, ainsi que les caches à correspondance directe avec un degré d'associativité égal à un. Finalement, nous supposons que la taille des lignes de cache d'un niveau doit être supérieure ou égale à celle des lignes de cache du niveau précédent et nous considérons dans un premier temps que chaque niveau de cache met en œuvre la politique de remplacement de cache LRU. Cette considération sera élargie à d'autres politiques de remplacement de cache dans la section 2.6 de ce chapitre.

2.1.3 Organisation du chapitre

La suite de ce chapitre est organisée de la manière suivante. Nous commençons par étudier la méthode proposée dans [68] pour analyser des hiérarchies de caches d'instructions non-inclusives (section 2.2) et nous verrons que cette méthode peut mettre en défaut la propriété de sûreté de l'estimation du temps d'exécution pire

cas. Nous proposerons ensuite des méthodes d'analyse statique sûres afin d'analyser des hiérarchies de caches d'instructions disposant des politiques de gestion non-inclusives (section 2.3), inclusives (section 2.4) et exclusives (section 2.5) avec une politique de remplacement de cache LRU pour tous les niveaux de la hiérarchie. Nous verrons ensuite comment étendre l'analyse à différentes politiques de remplacement de cache (section 2.6) et finalement comment prendre en compte la contribution des accès mémoire aux pire temps d'exécution (section 2.7). Une étude expérimentale sera ensuite présentée (section 2.8) en comparant les différentes politiques de gestion avant de conclure et d'aborder les perspectives de ces méthodes d'analyse.

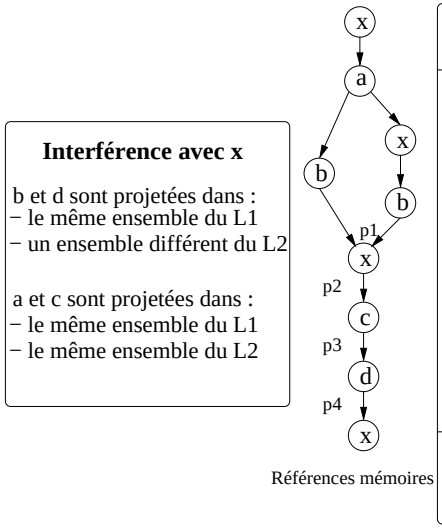
2.2 Approche existante et limitation

La première approche visant à analyser statiquement des hiérarchies de caches a été proposée dans [68] pour des hiérarchies de caches d'instructions non-inclusives utilisant la politique de remplacement LRU. Cette méthode vise à déterminer, pour chacune des références mémoire, le comportement pire cas, dans chaque niveau de la hiérarchie.

Pour ce faire, une simple analyse de chacun des niveaux de caches, par une analyse existante, n'est pas suffisante dû au filtrage se produisant entre les différents niveaux. En effet, à l'exécution lors d'une référence à la mémoire, l'accès est tout d'abord transmis au cache de premier niveau. Si l'accès produit un succès, l'information est directement retournée au processeur, sans aucune propagation de l'accès au niveau suivant. Si par contre, l'accès produit un défaut, l'accès est alors propagé au cache de niveau inférieur qui réalise le même type de traitement. L'accès peut ainsi être propagé jusqu'à la mémoire principale.

La solution proposée dans [68] pour prendre en compte les effets du filtrage consiste à analyser de façon séquentielle chaque niveau de cache, en se basant sur une analyse mono-niveau existante [4]. Cette analyse commence par le cache de premier niveau et se poursuit tout au long de la hiérarchie. La classification de comportement résultant de l'analyse d'un niveau leur permet de modéliser le filtrage et ainsi déterminer les références devant être prises en compte lors de l'analyse du niveau suivant. Plus précisément, les accès classifiés *Always-Hit* ne sont pas considérés comme propagés, lors de l'analyse, dans les niveaux suivants tandis que tous les autres le sont, ce qui revient à considérer que le pire cas se produit lorsque ces accès sont propagés.

Bien qu'intuitivement cette modélisation du filtrage semble correcte, nous allons voir au travers d'un exemple qu'elle s'avère poser des problèmes de sûreté lors de l'analyse de caches associatifs par ensembles. La figure 2.1 présente un graphe de flot de contrôle où chacun des blocs de base accède à une référence



	Cache abstrait du L1 et du L2	Classification de comportement	Contenu concret du L1 et du L2 sur le pire chemin	Comportement dans le L1 et le L2
p1	L1 {b} {a,x} L2 {a} {x}	x L1 always−miss L2 always−hit	L1 b x L2 a x	x L1 succès L2 pas d'accès
p2	L1 {x} {b} L2 {x} {a}	c L1 always−miss L2 always−miss	L1 x b L2 a x	c L1 défaut L2 défaut
p3	L1 {c} {x} L2 {c} {x}	d L1 always−miss L2 always−miss	L1 c x L2 c a	d L1 défaut L2 défaut
p4	L1 {d} {c} L2 {c} {x}	x L1 always−miss L2 always−hit	L1 d c L2 c a	x L1 défaut L2 défaut
Résultats des accès à x entre p1 et p4	p1: défaut L1 + succès L2 p4: défaut L1 + succès L2 Total: 2 défauts L1 + 2 succès L2		p1: succès L1 + no d'accès L2 p4: défaut L1 + défaut L2 Total: 1 succès + 1 défaut L1 + 1 défaut L2	

FIG. 2.1: Exemple du problème de sûreté de l'analyse de hiérarchies de *caches* non-inclusives existante [68]. Le problème de sûreté est observé sur le dernier accès à la référence mémoire x. Lors de l'analyse (partie gauche du tableau), l'accès est considéré comme produisant un succès dans le cache L2 tandis que lors de l'exécution dans le pire cas (partie droite du tableau), cet accès produit un défaut au niveau du cache L2.

mémoire. La hiérarchie de caches considérée est constituée de deux niveaux de caches (L1 et L2), chacun avec un degré d'associativité de deux et utilisant une politique de remplacement LRU. Le problème de sûreté est observé avec la référence x. Les autres références a, b, c et d ne posent pas de problème de sûreté car elles produisent tant lors de l'analyse qu'à l'exécution des défauts de cache dans les niveaux de cache L1 et L2. Elles sont introduites uniquement pour illustrer le problème de sûreté se produisant sur la référence x. Nous considérons dans cet exemple que les références :

- a et c sont projetées dans le même ensemble que x dans le cache L1 et le cache L2 ;
- b et d sont projetées dans le même ensemble que x dans le cache L1 et dans un ensemble différent dans le cache L2. Ce cas est courant car la taille du cache L1 est généralement plus petite que celle du cache L2.

La partie de gauche du tableau représente le contenu des états abstraits de cache suite à l'analyse statique aux points de programme $p1$, $p2$, $p3$ et $p4$. Pour simplifier, nous représentons uniquement l'ensemble du L1 et du L2 où la référence x est projetée. Dans l'exemple, $\boxed{\{u\} \mid \{v, w\}}$ représente le contenu possible de chacun des blocs de cache de l'ensemble en un point de programme. Le bloc de cache de gauche a un âge inférieur à celui de droite et $\{v, w\}$ signifie que les

deux références mémoire v et w, contenues dans deux lignes de cache différentes, peuvent être présentes dans le bloc de cache. La partie droite du tableau présente le contenu concret des caches aux mêmes points de programme lorsque que le pire chemin d'exécution est emprunté, soit dans notre exemple la partie droite de la conditionnelle.

Les contenus des états abstraits de cache du cache L1 et L2 à l'entrée de la conditionnelle, soit après l'accès à la référence a, sont identiques et ont pour valeur $\boxed{\{a\} \mid \{x\}}$. La branche de gauche de la conditionnelle modifie l'état abstrait du cache L1 en $\boxed{\{b\} \mid \{a\}}$ tandis que l'état abstrait de cache du cache L2 est lui inchangé car la référence b est projetée dans un autre ensemble du cache. La branche de droite, quant à elle, réalise tout d'abord un accès à la référence x, ce qui modifie l'état abstrait de cache du cache L1 en $\boxed{\{x\} \mid \{a\}}$ et l'état abstrait de cache du cache L2 reste inchangé car l'accès n'est pas propagé à ce niveau en raison de la présence de x dans le cache L1 (classification *Always-Hit* au niveau du cache L1). Puis, l'accès à la référence b modifie l'état abstrait de cache du cache L1 en $\boxed{\{b\} \mid \{x\}}$ tandis que l'état abstrait de cache du cache L2 est inchangé car la référence b est projetée dans un autre ensemble du cache. Au point de reconvergence de la conditionnelle (en $p1$), l'union des différents états abstraits de cache est réalisée et il en résulte un état abstrait de cache du cache L1 égal à $\boxed{\{b\} \mid \{a, x\}}$ et du cache L2 égal à $\boxed{\{a\} \mid \{x\}}$. L'accès à x en $p1$ est classifié *Always-Miss* au niveau du L1 (le nombre de références en conflit avec x contenues dans l'état abstrait de cache du cache L1 est égal au degré d'associativité). Par conséquent, l'accès est propagé au cache L2 et l'âge de x dans le cache L2 est mis à jour. Avec ce procédé, la référence x est présente dans l'état abstrait de cache du cache L2 en $p4$ et le nombre de conflits avec x en ce point est strictement inférieur au degré d'associativité donc la référence x est classifiée *Always-Hit* au niveau du cache L2. Avec cette classification entre $p1$ et $p4$, la contribution de la hiérarchie de caches au pire temps d'exécution pour la référence x est de deux défauts dans le cache L1 et deux succès dans le cache L2, comme résumé en partie basse du tableau.

Si nous regardons maintenant ce qu'il se passe au niveau du cache L1 et du cache L2 pour la référence x lors de l'exécution le long du pire chemin, nous observons que la contribution de la hiérarchie de caches au pire temps d'exécution pour la référence x est cette fois-ci de un succès dans le cache L1, de un défaut dans le cache L1 et de un défaut dans le cache L2.

En considérant une architecture où un défaut de cache est le pire cas et où le temps d'accès entre un succès (Ts) et un défaut (Td) dans le L2 est tel que : $2 * Ts_{L2} < Td_{L2}$, la valeur de la contribution de la hiérarchie de cache au pire temps d'exécution pour la référence x est sous estimée par l'analyse statique.

Le problème de sûreté, mis en avant dans cet exemple, est lié aux caractéris-

tiques de la chaîne d'accès ainsi qu'au fait de considérer des accès pouvant être filtrés sur certains chemins d'exécution comme des accès toujours propagés. Pour caractériser ce problème de sûreté plus formellement, il nous faut introduire la notion de distance de réutilisation dans un ensemble.

Définition 2.1 *La distance de réutilisation dans un ensemble d'une ligne de cache cl projetée dans l'ensemble e d'un cache de niveau ℓ est égale au nombre de lignes de cache distinctes projetées dans e et accédées entre deux accès à la ligne cl au niveau ℓ pour une séquence d'accès donnée.*

Par exemple, sur la figure 2.1, la distance de réutilisation dans un ensemble de la référence x au point $p4$ estimée lors de l'analyse statique est de 2 pour le L1 (égale au degré d'associativité et donc x est absent du cache L1) et de 1 pour le cache L2 (inférieure au degré d'associativité et donc x est présente dans la seconde voie). Si maintenant nous regardons au même point de programme la distance de réutilisation dans un ensemble de la référence x, lors de l'exécution le long du pire chemin, cette valeur est de 2 dans le L1 (identique à l'analyse statique) et également de 2 dans le L2 (supérieure à l'analyse statique et source du problème de sûreté). En résumé, l'analyse statique sous-estime dans cet exemple la distance de réutilisation dans un ensemble de la référence x.

La propagation systématique des accès à succès non garantis, que nous appellerons *incertain* par la suite, est à l'origine de la sous-estimation de la distance de réutilisation dans un ensemble. Cette sous-estimation peut conduire l'analyse statique à déterminer plus de succès de cache dans le niveau suivant, comparativement à l'exécution dans le pire cas.

Nous verrons dans la suite de ce chapitre comment détecter et prendre en compte les accès incertains lors de l'analyse et ainsi garantir la sûreté de l'estimation du pire temps d'exécution.

2.3 Analyse statique des hiérarchies de caches non-inclusives

Dans cette partie, nous nous intéressons à l'analyse des hiérarchies de caches non-inclusives. Plus précisément, nous considérons des hiérarchies vérifiant les quatre propriétés suivantes :

P1. Une information est recherchée dans le cache de niveau ℓ si et seulement si un défaut de cache se produit dans le cache de niveau $\ell - 1$ sauf pour le cache de premier niveau qui est toujours accédé ;

P2. L'âge des lignes de cache contenues dans le niveau ℓ est mis à jour, en fonction de la politique de remplacement de cache, à chaque fois qu'un accès se produit au niveau ℓ ;

P3. À chaque fois qu'un défaut de cache se produit au niveau ℓ, la ligne de cache contenant l'information provoquant ce défaut est intégralement chargée dans le cache de niveau ℓ ;

P4. Il n'y a aucune autre action sur le contenu des caches (i.e. recherches, modifications, invalidations...) autres que celles mentionnées ci-dessus.

2.3.1 Vue d'ensemble de l'analyse

Tout comme [68], notre analyse statique des hiérarchies de caches non-inclusives s'applique sur chaque niveau de cache de façon séquentielle en parcourant la hiérarchie à partir du cache de premier niveau. L'approche consiste à analyser le cache de premier niveau afin de déterminer le comportement pire cas dans ce niveau, pour toutes les références mémoire. À la suite de cette analyse, une classification de comportement (nommée CDC et constituée des classifications *Always-Hit* (AH), *Always-Miss* (AM), *First-Miss* (FM) et *Not Classified* (NC) définies dans [34, 101] et présentée dans le chapitre 1, section 1.4.4) pour le premier niveau de cache est associée à chaque référence mémoire. Cependant, comme nous avons pu le remarquer précédemment, cette classification seule ne permet pas de modéliser simplement et de façon sûre le comportement pire cas des accès mémoire dans l'ensemble de la hiérarchie en raison des accès incertains. En d'autres termes, elle ne permet pas en général de déterminer si un accès à une référence mémoire peut se produire ou non dans le niveau suivant.

Afin de prendre en compte le comportement induit par les accès incertains, nous introduisons une nouvelle classification : la classification d'accès au cache (CAC). Celle-ci permet de déterminer si un accès est garanti de se produire, de ne pas se produire ou encore s'il est incertain dans un niveau de cache donné. La combinaison de la classification de comportement et de la classification d'accès au cache d'un niveau est utilisée comme une entrée de l'analyse de cache du niveau suivant dans la hiérarchie mémoire. Une fois que tous les niveaux ont été analysés, la classification de comportement de chaque niveau est utilisée pour estimer le pire temps d'exécution. La figure 2.2 illustre la structure de notre analyse de la hiérarchie mémoire. Le symbole $\bigoplus$ représente la fonction combinant la classification de comportement avec la classification d'accès au cache dont le résultat est un paramètre de l'analyse du niveau inférieur.

2.3.2 La classification d'accès au cache (CAC)

Afin de déterminer si une référence mémoire accède à un niveau de cache donné, nous introduisons le concept de *classification d'accès au cache* (CAC). Cette classification correspond à une formalisation du filtrage des accès et est utilisée par la suite comme une entrée de l'analyse de cache de chaque niveau

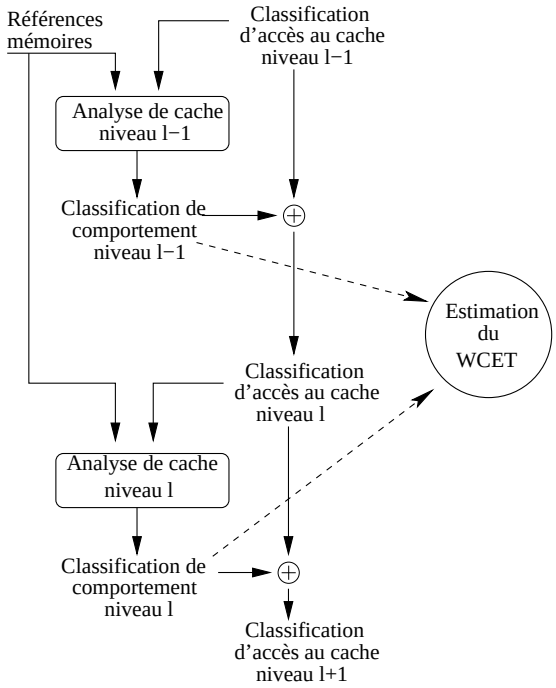


FIG. 2.2: Vue d'ensemble de l'analyse statique de hiérarchies de caches non-inclusives.

pour décider si l'accès doit être pris en compte ou non lors de l'analyse. Elle est associée à chaque référence mémoire et ce pour chaque niveau de cache. Elle est composée des quatre éléments de classification suivants :

- *Never* (N) : l'accès à la référence mémoire ne se produit jamais à ce niveau de cache ;
- *Always* (A) : l'accès à la référence mémoire se produit toujours à ce niveau de cache ;
- *Uncertain-Never* (U-N) : l'accès à la référence mémoire peut se produire ou non lors du premier accès mais les accès suivants ne se produisent jamais à ce niveau de cache ;
- *Uncertain* (U) : l'accès à la référence mémoire peut se produire mais sans certitude à ce niveau de cache.

La classification d'accès au cache pour une référence r au niveau de cache ℓ (noté $CAC_{r,\ell}$) dépend d'une part du filtrage réalisé par le niveau de cache précédent, information obtenue par la classification de comportement de la référence r au niveau $\ell - 1$ (noté $CDC_{r,\ell-1}$) ainsi que le filtrage réalisé par l'ensemble des niveaux précédents le niveau $\ell - 1$, information véhiculée par la classification d'accès au cache du niveau $\ell - 1$ ($CAC_{r,\ell-1}$).

Pour le cas particulier du premier niveau de la hiérarchie, la classification d'accès au cache est toujours égale à *Always* car tous les accès accèdent le cache de premier niveau. L'autre cas, occasionnant une classification *Always*, se produit quand la $CAC_{r,\ell-1}$ et la $CDC_{r,\ell-1}$ sont respectivement égales à *Always* et *Always-Miss*. Dans cette configuration, nous avons la garantie que l'accès à la référence r provoquera toujours un accès au cache de niveau ℓ.

Inversement, $CAC_{r,\ell}$ est égale à *Never* lorsque la $CDC_{r,\ell-1}$ est *Always-Hit*. En d'autres termes, nous avons la garantie que l'accès à la référence r ne produira pas d'accès au niveau ℓ car un succès de cache est garanti dans le niveau $\ell - 1$. La seconde possibilité pour avoir $CAC_{r,\ell}$ égale à *Never* se produit quand $CAC_{r,\ell-1}$ est également égale à *Never*, ce qui signifie qu'une classification *Always-Hit* est présente dans un des niveaux précédents.

L'ensemble des autres combinaisons entre la $CAC_{r,\ell-1}$ et la $CDC_{r,\ell-1}$ produit soit une classification d'accès *Uncertain* soit *Uncertain-Never* exprimant ainsi l'incertitude que l'accès s'effectue au cache de niveau ℓ et permet de ce fait de capturer la notion d'accès incertain. Cette information est utilisée par la suite pour modéliser le comportement des accès incertains lors de l'analyse de cache de chaque niveau en explorant conjointement le cas où l'accès est propagé et celui où il ne l'est pas afin de garantir la sûreté de l'analyse.

Concernant la classification d'accès *Uncertain-Never*, elle représente un cas particulier de la classification *Uncertain* induit par la détection d'une classification de comportement *First-Miss* dans un niveau précédent de la hiérarchie de caches. La présence d'une classification *First-Miss* dans un niveau précédent est

ainsi conservée au sein de la classification d'accès sans surcoût supplémentaire. Cette information n'est pas directement utilisée par notre analyse de hiérarchie de caches, elle sera utilisée ensuite par les analyses, portant sur les architectures multi-cœur, présentées dans le chapitre 3.

L'ensemble des cas possibles pour la classification d'accès au cache du niveau ℓ en fonction des classifications du niveau précédent ($CAC_{r,\ell-1}$ et $CDC_{r,\ell-1}$) est synthétisé dans le tableau 2.1.

$CAC_{r,\ell-1}$ \ $CDC_{r,\ell-1}$	AM	AH	FM	NC
A	A	N	U-N	U
U	U	N	U-N	U
U-N	U-N	N	U-N	U-N
N	N	N	N	N

TAB. 2.1: Classification d'accès au cache de niveau ℓ ($CAC_{r,\ell}$)

Le contenu du tableau motive le besoin de la classification d'accès au cache. En effet, dans le cas d'un *Always-Miss* au niveau $\ell-1$, déterminer si l'accès doit être considéré au niveau ℓ requiert plus d'informations que ce que peut fournir la classification de comportement. Si la référence est toujours propagée au niveau $\ell-1$ ($CAC_{r,\ell-1} = A$) alors elle doit également l'être au niveau ℓ, tandis que si la référence est incertaine au niveau $\ell-1$ ($CAC_{r,\ell-1} = U$ ou $CAC_{r,\ell-1} = U-N$), elle doit dans ce cas également être considérée comme incertaine au niveau ℓ.

2.3.3 Prise en compte de la classification d'accès lors de l'analyse d'un niveau de cache

L'introduction de la classification d'accès au cache amène une nouvelle dimension aux analyses de caches existantes [69, 101] opérant sur un niveau de cache unique. Dans ces analyses, la notion d'accès incertains n'avait nullement besoin d'exister, car le cache de premier niveau est accédé lors de chaque accès. Par contre, pour analyser les niveaux inférieurs de la hiérarchie, prendre en compte les accès incertains devient une nécessité vis-à-vis de la sûreté de l'estimation du pire temps d'exécution.

Afin de les prendre en compte lors de l'analyse, notre approche consiste à étendre une analyse de cache mono-niveau existante en y incorporant les informations véhiculées par la classification d'accès au cache. Nous nous basons sur l'analyse mono-niveau [101] détaillée précédemment (chapitre 1, section 1.4.4) pour ces propriétés mathématiques liées à l'utilisation de l'interprétation abstraite [25, 26]. Pour rappel, cette dernière définit trois analyses statiques : *Must*, *May* et *Persistence* afin de déterminer si une référence est assurément présente

dans le cache, peut être présente dans le cache ou bien si elle est persistante dans le cache après y avoir été préalablement chargée.

Chaque analyse définit, en accord avec sa sémantique, deux fonctions nommées *Update* et *Join*. La fonction *Update* modélise la mise à jour des informations contenues dans le cache suite à un accès. Elle prend en entrée la référence mémoire accédée et l'état du cache abstrait modélisant le contenu du cache avant l'accès (ACS_{in}) et retourne l'état de cache abstrait (ACS_{out}) représentant l'état du cache après l'accès. La fonction *Join* quant à elle, permet de fusionner deux états de caches abstraits distincts en un seul lorsqu'un bloc de base à plus d'un prédécesseur, comme par exemple au point de re-convergence d'une structure conditionnelle. Pour l'analyse *Must*, elle conserve l'âge maximal de chaque référence mémoire présente dans les deux états abstraits de caches. Pour l'analyse *Persistence*, elle conserve l'âge maximal de chaque référence présente dans au moins un des deux états abstraits de caches. Réciproquement pour l'analyse *May*, elle conserve l'âge minimal de chaque référence présente dans au moins un des deux états de caches abstraits.

L'extension de [101] au cas des hiérarchies de caches nécessite de prendre en compte les informations fournies par la classification d'accès au cache. Celles-ci concernant les accès au cache, il est naturel de prendre en compte cette classification lors de chaque appel à la fonction *Update*. Pour ce faire, nous redéfinissons cette fonction, que nous appellerons $Update_m$ par la suite, pour la différencier de la fonction originale, pour chaque classification d'accès au cache.

- ***Always.*** Pour cette classification, l'accès à la référence r se produit toujours et de ce fait la fonction *Update* d'origine peut être directement utilisée.

$$ACS_{out} = Update(ACS_{in}, r) \text{ donc, } Update_m \Leftrightarrow Update$$

- ***Never.*** Pour cette classification, l'accès à la référence r ne se produit jamais. Ce type d'accès doit donc être ignoré lors de l'analyse :

$$ACS_{out} = ACS_{in} \text{ donc, } Update_m \Leftrightarrow \text{fonction identité}$$

- ***Uncertain* ou *Uncertain-Never.*** Pour ces classifications reflétant le cas des accès incertains, l'analyse doit prendre en compte les deux comportements possibles produisant chacun un état abstrait de cache :
  - L'accès est effectué, le résultat est équivalent au cas des accès classifiés *Always* ;
  - L'accès n'est pas effectué, le résultat est équivalent au cas des accès classifiés *Never*.

  Pour obtenir l'état abstrait de cache résultant d'un accès incertain, nous fusionnons ensuite ces deux états abstraits de caches avec la fonction *Join*, comme illustré par la figure 2.3.

$$ACS_{out} = Join(Update(ACS_{in}, r), ACS_{in}) \text{ donc,}$$

$$Update_m(ACS_{in}, r) \Leftrightarrow Join(Update(ACS_{in}, r), ACS_{in})$$

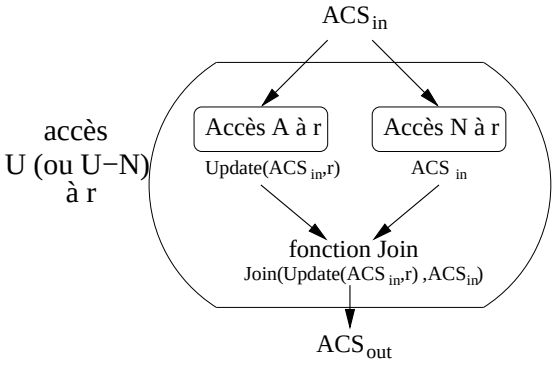


FIG. 2.3: Traitement des accès incertains. L'analyse prend en paramètre l'état de cache abstrait avant l'accès puis explore les deux possibilités : l'accès se produit (branche de gauche) et l'accès ne se produit pas (branche de droite) produisant alors deux états de caches abstraits distincts. Le traitement se poursuit en appliquant la fonction *Join* sur les deux états de cache abstraits servant à produire l'état de cache abstrait résultant de l'accès incertain.

L'utilisation des fonctions *Update* et *Join* initiales, lors de la définition de la fonction $Update_m$, permet de s'abstraire de la sémantique de chacune des analyses, celle-ci étant assurée par les fonctions initiales.

2.3.4 Sûreté de l'analyse

Concernant la sûreté de la classification de comportement, celle-ci est garantie de façon évidente pour les accès de type *Always* par la fonction *Update* initiale et pour les accès de type *Never* car ils ne modifient pas l'état abstrait de cache. Pour les accès incertains, la sûreté est également assurée grâce à la conservation de la sémantique de chacune des analyses. Pour les analyses *Must* et *Persistence*, la fonction $Update_m$ maintient bien l'âge maximal dans l'état abstrait de cache de chaque référence mémoire par l'utilisation de la fonction *Join* appliquée sur les deux états abstraits de caches résultant du cas où l'accès se produit et de celui où il ne se produit pas. De façon similaire, pour l'analyse *May*, c'est l'âge minimal qui est maintenu. La sûreté est ainsi garantie par la fonction $Update_m$ pour chacun des différents types d'accès et ce pour chacune des analyses.

2.3.5 Terminaison de l'analyse

Les auteurs de [101] montrent que le domaine des états de cache abstraits est fini et que les fonctions *Update* et *Join* sont monotones pour chacune des analyses. La condition des chaînes ascendantes (dans un domaine fini, toutes les chaînes ascendantes sont finies) leur permet alors de prouver la terminaison du calcul de point fixe de chaque analyse.

Dans notre approche, l'unique modification par rapport à [101] pour l'analyse d'un niveau de cache porte sur la fonction *Update* avec la prise en compte de la classification d'accès au cache. Pour montrer la terminaison de notre analyse, il nous suffit donc de montrer que la fonction $Update_m$ est monotone.

Preuve : Chaque référence mémoire à une classification d'accès au cache constante pour un niveau donné. Il nous faut donc montrer que la fonction $Update_m$ est monotone pour chaque classification d'accès :

- Classification *Always* : la fonction $Update_m$ est équivalente à la fonction *Update* donc elle est monotone ;
- Classification *Never* : la fonction $Update_m$ est équivalente à la fonction identité donc elle est monotone ;
- Classification *Uncertain* ou *Uncertain-Never* : la fonction $Update_m$ est une composition des fonctions *Update* et *Join*. Comme la composition de fonctions monotones est également monotone, $Update_m$ est monotone.

□

2.3.6 Exemple

Reprenons l'exemple étudié précédemment, montrant le problème de sûreté lié aux accès incertains (section 2.2), en y appliquant cette fois notre analyse de cache multi-niveaux. Le tableau de la figure 2.4 présente pour chacun des deux niveaux de cache :

- la classification d'accès au cache (CAC) de chaque référence mémoire ;
- le contenu des états abstraits de cache, une fois le point-fixe atteint, avant et après avoir effectué l'accès à la référence mémoire et ce pour les analyses *May* et *Must* ;
- la classification de comportement de chaque référence mémoire.

Au point de programme $p1$, la référence mémoire x n'est ni dans l'état abstrait de cache de l'analyse *May*, ni dans celui de l'analyse *Must* du L1. La classification de comportement est donc *Always-Miss* et la classification d'accès au cache L2 est *Always* produisant alors un accès systématique au L2.

Au point $p2$, la référence x est présente dans l'état abstrait de cache de l'analyse *Must*, amenant une classification de comportement *Always-Hit* dans le L1 et une classification *Never* pour l'accès au L2, ce qui a pour effet de ne pas modifier le contenu des l'états abstraits de cache du L2.

Au point $p3$, source du problème de sûreté, la référence x est présente uniquement dans l'état abstrait de cache de l'analyse *May* du cache L1. La classification de comportement vis-à-vis du L1 est donc *Not Classified* et la classification d'accès au cache L2 est *Uncertain*. Au niveau du L2, la référence x est présente dans l'état de cache abstrait de l'analyse *Must* donnant ainsi la classification de comportement *Always-Hit*. Regardons plus précisément la mise à jour des états de cache abstraits du L2 provoquée par la classification *Uncertain*. Pour l'analyse *May*, la fonction *Join* est appliquée entre $\boxed{\{a\} \mid \{x\}}$ (l'état si l'accès ne se produit pas) et $\boxed{\{x\} \mid \{a\}}$ (l'état si l'accès se produit) donnant alors l'état $\boxed{\{x, a\} \mid \{\}}$. Pour l'analyse *Must*, les états de cache abstraits, si l'accès se produit ou non, sont identiques à l'analyse *May*, ce qui donne en résultat $Join(\boxed{\{a\} \mid \{x\}}, \boxed{\{x\} \mid \{a\}}) = \boxed{\{\} \mid \{a, x\}}$.

Nous pouvons remarquer que pour l'analyse *May*, déterminant si une référence mémoire peut être présente dans un niveau de cache, l'âge de la référence x est mis à jour sans modifier l'âge des autres références mémoire, assurant ainsi la conservation d'un âge minimal lors de l'analyse (à l'exécution l'âge est supérieur ou égal à celui de l'analyse *May*). Réciproquement, pour l'analyse *Must*, la conservation de l'âge maximal est assurée en présence d'accès incertains. La distance de réutilisation n'est donc pas sous-estimée pour les analyses conservant l'âge maximal et n'est pas sur-estimée pour les analyses conservant l'âge minimal.

L'étude expérimentale concernant cette analyse est réalisée à la fin du chapitre (section 2.8) afin d'effectuer une comparaison entre les différentes politiques de gestion de hiérarchie de caches.

2.3.7 Analyse indépendante des différents niveaux de caches de la hiérarchie mémoire

L'analyse des hiérarchies de caches non-inclusives telle qu'elle est présentée ci-dessus, est appliquée de façon séquentielle sur tous les niveaux de caches de la hiérarchie. Le parcours débute par le premier niveau de cache et le résultat de l'analyse de ce dernier est nécessaire pour déterminer la classification d'accès au cache et assurer ainsi la sûreté de l'analyse du niveau suivant. La dépendance entre un niveau de cache et le suivant est donc uniquement dû à la modélisation du filtrage de chaque accès, information véhiculée par la classification d'accès au cache.

Nous avons introduit cette classification afin de capturer la notion d'accès incertain par l'intermédiaire de la classification *Uncertain*. Les autres éléments de cette classification (*Always*, *Never* et *Uncertain-Never*) peuvent être vus comme un raffinement de cette dernière. Prenons par exemple la classification *Always*, sa sémantique indique que l'accès à toujours lieu dans un niveau de cache donné, ce

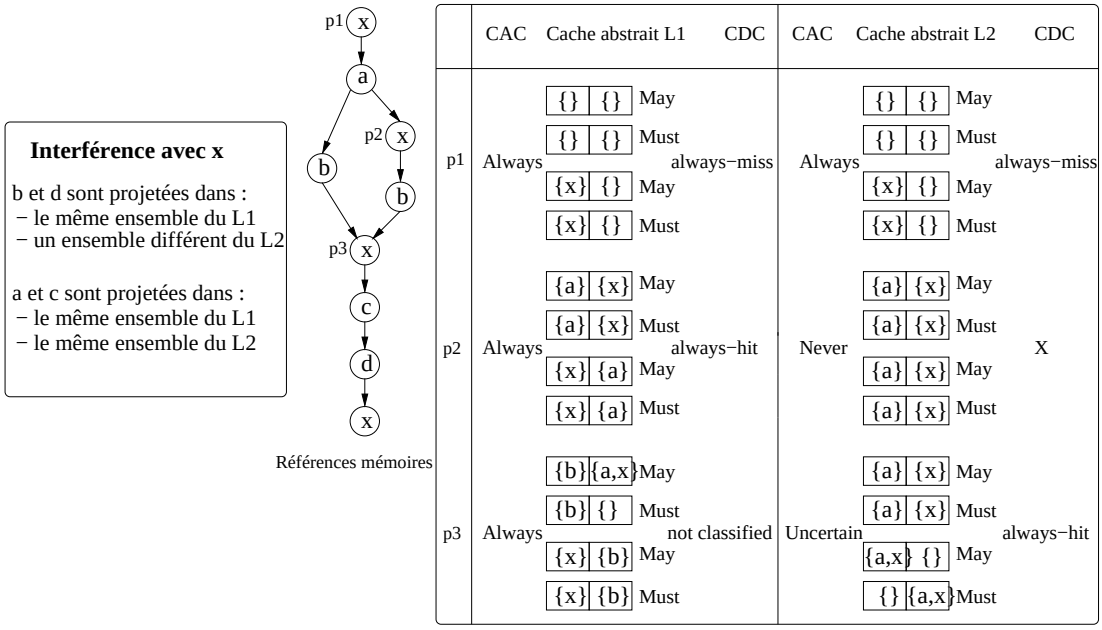


FIG. 2.4: Exemple du résultat de l'analyse statique des hiérarchies de caches non-inclusives. Cet exemple montre le résultat du traitement des accès à la référence x classifiés *Always* ($p1$), *Never* ($p2$) et *Uncertain* ($p3$), lors de l'analyse *Must* et *May*.

qui est bien sûr plus précis que celle des accès classifiés *Uncertain* indiquant que l'accès *peut* avoir lieu. Cependant, un accès ayant toujours lieu peut être considéré comme un accès pouvant avoir lieu lors de l'analyse de cache sans mettre en défaut sa sûreté.

Nous avons bien la classification *Always* qui est un raffinement de la classification *Uncertain* notée $Always \prec Uncertain$ et signifiant que les accès classifiés *Always* peuvent être interprétés comme des accès classifiés *Uncertain* sans affecter la sûreté de l'analyse. En suivant un raisonnement similaire pour les autres classifications, nous obtenons : $Never \prec Uncertain$ et $Uncertain - Never \prec Uncertain$.

Dès lors, en considérant tous les accès mémoire comme ayant une classification *Uncertain*, pour tous les niveaux de cache à l'exception du premier niveau, l'analyse de chaque niveau de cache devient indépendante du résultat de l'analyse des niveaux précédents sans remettre en cause la sûreté de l'analyse hiérarchique.

Évidemment, considérer l'ensemble des accès à un niveau comme ayant une classification *Uncertain* dégrade potentiellement la précision de l'analyse dû à la perte d'information sur les références mémoire accédant effectivement ce niveau de cache. Néanmoins, l'intérêt principal de cette indépendance, pour l'analyse des hiérarchies de caches non-inclusives, est la possibilité de pouvoir effectuer l'analyse de chaque niveau en parallèle permettant ainsi de réduire de façon significative le temps de calcul nécessaire à l'analyse.

Regardons maintenant plus en détail l'impact qualitatif de cette considération ($CAC = U$) sur chacune des analyses pour un cache de niveau $\ell \geq 2$:

- **analyse *May*.** Cette analyse permet de déterminer si une référence mémoire est absente du cache (*Always-Miss*) ou si elle peut être présente (*Not Classified*). Comme chaque accès est classifié *Uncertain* et que l'analyse *May* conserve l'âge minimal dans le cache de chacune des références, le résultat de l'analyse produit une classification *Not Classified* pour chaque référence, excepté pour le premier accès à une ligne de cache où la classification sera *Always-Miss*. Le premier accès à une ligne de cache peut, sans remettre en cause la sûreté de l'analyse, être classifié *Not Classified* ce qui permet d'éviter l'analyse *May* lors d'une analyse indépendante des différents niveaux.
- **analyse *Must*.** Cette analyse détermine si une référence mémoire est toujours présente dans le cache (*Always-Hit*). Afin de mieux visualiser le traitement d'une référence mémoire classifiée *Uncertain*, prenons un état de cache abstrait initialement vide et un accès mémoire à la référence x. Lors du traitement de cet accès, la fonction $Update_m$ applique la fonction $Join$ entre $\boxed{\{\} \mid \{\}}$ (l'état de cache abstrait si l'accès n'est pas réalisé) et $\boxed{\{x\} \mid \{\}}$ (l'état de cache abstrait si l'accès est réalisé). Pour l'analyse *Must*, la fonction $Join$ maintient l'âge maximal d'une référence mé-

moire présente dans les deux états de cache abstraits. Dans notre cas, $Update_m = Join(\boxed{\{\} \mid \{\}}, \boxed{\{x\} \mid \{\}}) = \boxed{\{\} \mid \{\}}$. L'analyse *Must* devient de ce fait inexploitable car elle ne conserve aucune référence dans les états de cache abstraits ne pouvant ainsi classifiée aucune référence mémoire comme *Always-Hit*.
La perte de cette information, pour les niveaux de caches autres que le premier niveau, a un impact minime sur la précision de l'analyse. En effet l'analyse *Must* détecte principalement la localité spatiale liée aux accès successifs à des références mémoire contenues dans une même ligne de cache. Or, si la taille des lignes de cache des différents niveaux est la même, l'analyse *Must* du cache de premier niveau (où tous les accès sont classifiés *Always* pour le premier niveau) détecte déjà l'intégralité de la localité spatiale capturée par la hiérarchie de caches.
Si la taille des lignes de cache des niveaux suivants est plus grande, prenons par exemple deux fois plus grandes (deux lignes de cache du premier niveau sont regroupées dans une ligne de cache du second niveau), l'accès à une référence contenue dans la première ligne garantit ensuite la présence dans le second niveau de la ligne de cache. L'accès à la deuxième ligne produit un défaut dans de le cache de premier niveau et un succès dans le cache de second niveau grâce à la régularité des accès aux instructions. L'analyse est par contre dans l'incapacité de capturer ce succès à cause de la classification *Uncertain*. Ce pessimisme peut être en partie compensé par un post traitement intra bloc de base changeant la classification de comportement des accès successifs (sauf le premier accès) à une même ligne de cache en *Always-Hit* lors de l'analyse d'architecture ne souffrant pas d'anomalie temporelle [61, 104, 87].

– **analyse *Persistence*.** Cette analyse détermine si une référence mémoire n'est pas évincée après avoir été préalablement chargée dans le cache (*First-Miss*). Cette fois encore, regardons le traitement de la fonction $Update_m$ basée sur la fonction *Join* afin de visualiser l'impact des références mémoire classifiées *Uncertain*. Reprenons le même exemple que précédemment, à savoir un état de cache abstrait vide et un accès à la référence mémoire x. Pour l'analyse *Persistence*, la fonction *Join* maintient l'âge maximal d'une référence mémoire présente dans au moins un des deux états de cache abstraits. Nous obtenons, dans cet exemple, $Update_m = Join(\boxed{\{\} \mid \{\}}\ \boxed{\{\}}, \boxed{\{x\} \mid \{\}}\ \boxed{\{\}}) = \boxed{\{x\} \mid \{\}}\ \boxed{\{\}}$. À la différence de l'analyse *Must*, la référence mémoire est contenue dans l'état de cache abstrait résultant permettant ainsi de détecter une partie de la localité temporelle capturée par le cache de chaque niveau. Prenons un second exemple où cette fois-ci l'état de cache abstrait contient déjà la référence x dans la seconde

voie. la fonction $Update_m$ devient alors :

$$Update_m = Join(\boxed{\{\} \mid \{x\}}\ \boxed{\{\}}, \boxed{\{x\} \mid \{\}}\ \boxed{\{\}}) = \boxed{\{\} \mid \{x\}}\ \boxed{\{\}}$$

Ce second exemple met en évidence le fait que l'âge d'une référence mémoire déjà présente dans l'état de cache abstrait n'est pas mis à jour lors d'un accès à une référence mémoire classifiée *Uncertain*. Ce comportement réduit la capacité à détecter la localité temporelle capturée par le cache de chaque niveau.

L'indépendance entre les niveaux de cache lors de l'analyse de la hiérarchie mémoire permet donc d'analyser chaque niveau en parallèle en appliquant uniquement les analyses *Must* et *Persistence* sur le cache de premier niveau et l'analyse *Persistence* sur les niveaux suivants. Bien que qualitativement, la perte de précision semble importante, nous verrons lors de l'évaluation de performance (section 2.8.2.2) que la perte réelle de précision peut s'avérer négligeable dans de nombreux cas, en fonction de la structure du code analysé et de la structure de la hiérarchie mémoire.

2.4 Analyse statique des hiérarchies de caches inclusives

2.4.1 Fonctionnement et propriétés

Le comportement des hiérarchies de caches inclusives est similaire aux hiérarchies de caches non-inclusives. Lors d'un accès à une référence mémoire, la ligne de cache contenant la référence mémoire est chargée dans tous les niveaux de cache où un défaut de cache se produit. Une hiérarchie de caches inclusive fournit tout de même une propriété supplémentaire entre les différents niveaux de caches de la hiérarchie : l'inclusion. Plus précisément, tout le contenu d'un niveau de cache est inclus dans le contenu de chacun des niveaux inférieurs de la hiérarchie.

Pour assurer la propriété d'inclusion, poser des hypothèses sur la structure des différents caches de la hiérarchie n'est pas une condition suffisante [7] pour des caches associatifs par ensembles disposant tous d'une politique de remplacement LRU. Afin d'illustrer ce propos, prenons la chaîne d'accès [a, b, a, c] ainsi que deux niveaux de caches disposant chacun d'un degré d'associativité de deux. Chacune des références mémoire est projetée dans le même ensemble dans les deux niveaux. La figure 2.5 présente la chaîne d'accès ainsi que le contenu des caches (concrets) avant et après chaque référence, en considérant initialement que les deux niveaux de caches sont vides. Le fait de référencer en alternance a permet son maintien dans le cache de premier niveau sans propager l'accès et donc sans mettre à jour son âge dans le second niveau. Ainsi, après l'accès à la référence c, la

référence a est évincée du second niveau alors qu'elle est toujours présente dans le premier niveau, mettant ainsi en défaut la propriété d'inclusion. Ce phénomène est reproductible pour des degrés d'associativité plus grands en augmentant le nombre d'alternances à la référence a montrant ainsi qu'aucune propriété sur la structure des caches associatifs par ensembles ne permet de garantir la propriété d'inclusion.

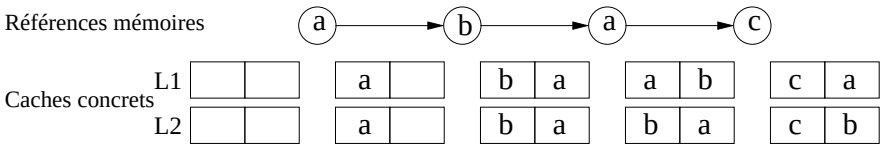


FIG. 2.5: Exemple du traitement d'une chaîne d'accès montrant que des hypothèses sur la structure de la hiérarchie de caches ne permettent pas de garantir la propriété d'inclusion. Après avoir effectué tous les accès, la référence a est présente dans le cache L1 mais ne l'est pas dans le cache L2.

Afin de maintenir l'inclusion, un mécanisme additionnel est alors requis par rapport aux hiérarchies de caches non-inclusives. Un procédé d'invalidation des lignes de caches est généralement employé pour maintenir cette propriété.

La figure 2.6 reprend l'exemple précédent en illustrant le mécanisme d'invalidation requis lors de l'accès à la référence mémoire c pour garantir l'inclusion. Cet exemple montre une méthode performante d'invalidation évitant la création de blocs de cache vides dans le cache de premier niveau. Pour ce faire, lorsque l'accès produit un défaut dans le cache de premier niveau, l'accès est propagé au second niveau. La référence mémoire étant également absente du second niveau, la ligne de cache, qui sera évincée lors du chargement de la référence c, est d'ores et déjà connue, grâce à la politique de remplacement, permettant ainsi de faire l'invalidation de cette ligne dans le niveau supérieur. La ligne de cache contenant la référence c peut alors prendre le bloc de cache libéré par l'invalidation dans le cache de premier niveau, évitant ainsi de laisser des blocs de cache vides issus du mécanisme d'invalidation.

Les hypothèses, précédemment introduites, sur le fonctionnement des hiérarchies de caches non-inclusives ne sont pas toutes directement utilisables pour les hiérarchies de caches inclusives dû à l'ajout du mécanisme d'invalidation. Nous considérons des hiérarchies de cache inclusives vérifiant les propriétés suivantes :

P1. Une information est recherchée dans le cache de niveau ℓ si et seulement si un défaut de cache se produit dans le cache de niveau $\ell - 1$ sauf pour le cache de premier niveau qui est toujours accédé ;

P2. L'âge des lignes de cache contenues dans le niveau ℓ est mis à jour, en fonction de la politique de remplacement de cache, à chaque fois qu'un accès est propagé au niveau ℓ ;

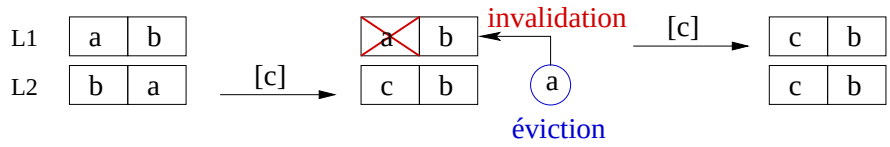


FIG. 2.6: Exemple du mécanisme d'invalidation servant à forcer l'inclusion lorsqu'une référence est évincée d'un niveau de cache. Sur cet exemple, la référence a est évincée du cache L2 lors de l'accès à la référence c. Le mécanisme d'invalidation libère le bloc de cache du L1 contenant cette référence, et permet ainsi d'y stocker la référence c tout en garantissant l'inclusion.

P3. A chaque fois qu'un défaut de cache se produit au niveau ℓ, la ligne de cache contenant l'information provoquant ce défaut est intégralement chargée dans le cache de niveau ℓ ;

P4. L'invalidation d'une ligne de cache cl au niveau ℓ se produit quand cl est évincée d'un niveau $\ell' > \ell$;

P5 Il n'y a aucune autre action sur le contenu des caches (i.e. recherches, modifications...) autres que celles mentionnées ci-dessus.

Les propriétés P1, P2 et P3 sont inchangées par rapport à celles définies pour les hiérarchies de caches non-inclusives tandis que les propriétés P4 et P5 sont une relaxation de la propriété P4 afin d'autoriser le mécanisme d'invalidation. Concernant la propriété P2, l'invalidation d'une ligne de cache dans un niveau ℓ n'est pas assimilable à un accès propagé dans le niveau ℓ et ne modifie donc pas l'âge des autres lignes de caches présentes dans ce niveaux.

2.4.2 Problématique de l'analyse

Par rapport à l'analyse des hiérarchies de caches non-inclusives, il nous faut en plus pour les hiérarchies de caches inclusives intégrer le mécanisme d'invalidation servant au maintien de la propriété d'inclusion. La prise en compte de ce mécanisme requiert la connaissance des lignes de cache potentiellement évincées en chaque point de programme par les différents niveaux de la hiérarchie. Cette information n'étant disponible qu'une fois tous les niveaux de cache analysés, nous proposons de prendre en compte les effets du mécanisme d'invalidation lors d'un post traitement, en modifiant la classification de comportement des accès aux lignes de cache potentiellement invalidées.

Cependant, cette modification de la classification de comportement pour un niveau de cache n'est pas sans effet sur l'analyse des niveaux suivants. En effet, nous avons vu précédemment que la classification d'accès au cache, servant à modéliser le filtrage des accès, est directement liée à la classification de comportement des accès mémoire.

Nous observons donc une dépendance circulaire entre d'une part, la classifica-

tion d'accès au cache utilisée pour analyser les niveaux suivants et d'autre part, le résultat de l'analyse des niveaux suivants nécessaire pour déterminer les lignes de cache à invalider dans les niveaux précédents de la hiérarchie. La prise en compte des invalidations modifie la classification de comportement des accès et donc par effet de bord la classification d'accès au cache des niveaux suivants.

Afin de casser cette dépendance circulaire, nous reprenons la méthode précédente (paragraphe 2.3.7) permettant de rendre l'analyse de chaque niveau indépendante en utilisant la classification d'accès au cache *Uncertain* pour toutes les références mémoire et ce pour tous les niveaux de cache à l'exception du premier niveau.

2.4.3 Vue d'ensemble de l'analyse

L'analyse se déroule en commençant par l'analyse de chacun des niveaux comme pour l'analyse des hiérarchies de caches non-inclusives en ignorant les invalidations. Le résultat de l'analyse de chaque niveau est utilisé par la suite pour détecter l'ensemble des lignes de cache potentiellement évincées en chaque point de programme, et ce pour chacun des niveaux à l'exception du premier niveau (ce dernier ne pouvant pas être à l'origine de l'invalidation d'une ligne de cache). La modélisation du procédé d'invalidation revient alors à modifier la classification de comportement des références mémoire contenues dans l'ensemble des lignes de cache potentiellement évincées des niveaux inférieurs. La figure 2.7 illustre la structure de notre analyse pour les hiérarchies de caches inclusives. Comparativement à l'analyse précédente, nous pouvons observer la présence du post traitement remontant les lignes de caches potentiellement évincées par les niveaux inférieurs de la hiérarchie (opérateur $\textcircled{U}$) afin de modifier la classification de comportement pour tenir compte des invalidations (opérateur $\textcircled{I}$).

2.4.4 Détection des lignes potentiellement évincées

L'ensemble des lignes de cache potentiellement évincées par un niveau de cache ℓ en un point de programme p, noté $PossiblyEvicted_{\ell,p}$, est une information produite lors de l'analyse de cache de chacun des niveaux par l'analyse *Persistence*. Cette analyse permet de déterminer si une ligne de cache ne sera pas évincée après avoir été préalablement chargée. La mise en œuvre de cette analyse utilise une voie virtuelle ajoutée à chaque ensemble de l'état de cache abstrait. Une fois l'analyse effectuée, cette voie contient l'ensemble des lignes de cache potentiellement évincées en un point de programme pour le niveau analysé. Par conséquent, la détection des lignes de cache potentiellement évincées ne nécessite pas la définition d'une nouvelle analyse, les états de cache abstraits de l'analyse *Persistence* sont suffisants pour calculer l'ensemble $PossiblyEvicted_{\ell,p}$.

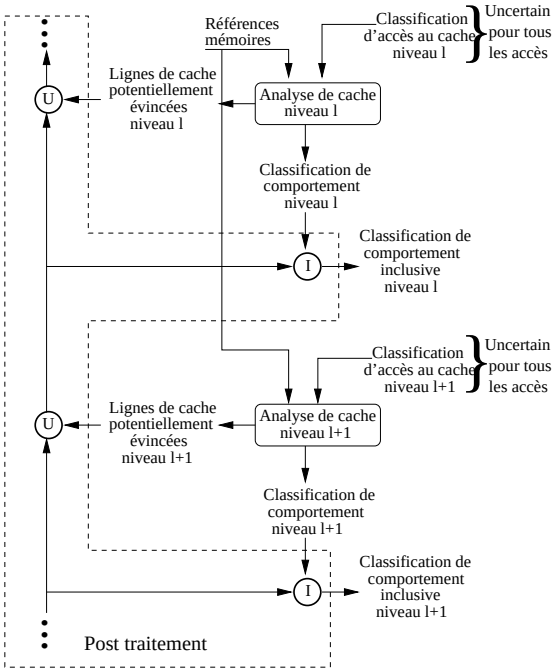


FIG. 2.7: Vue d'ensemble de l'analyse statique de hiérarchies de caches inclusives.

2.4.5 Modélisation du procédé d'invalidation : modification de la classification de comportement

L'analyse de cache de chacun des niveaux est effectuée sans prendre en compte les invalidations nécessaires au maintien de la propriété d'inclusion. La classification de comportement ainsi obtenue doit donc être modifiée, pour intégrer l'effet du mécanisme d'invalidation, en utilisant les lignes de cache potentiellement évincées par chacun des niveaux inférieurs de la hiérarchie.

Pour chaque niveau de cache ℓ, point de programme p et référence mémoire r contenue dans une ligne de cache cl, si la classification de comportement de r est *Always-Hit* ou *First-Miss* et que cl appartient à l'ensemble $PossiblyEvicted_{\ell',p}$ avec $\ell' > \ell$, alors la classification de comportement de r doit être changée en *Not Classified* représentant ainsi le fait que la ligne de cache est potentiellement invalidée avant l'accès à la référence r. La classification *Not Classified* est utilisée à la place de la classification *Always-Miss* pour garantir la sûreté de l'analyse car l'ensemble $PossiblyEvicted_{\ell',p}$ est un sur ensemble des lignes de cache pouvant être invalidées à l'exécution. Plus formellement, la classification de comportement de la référence mémoire r dans un niveau de cache ℓ est modifiée en *Not Classified* si :

$$(CDC_{r,\ell} = AH \vee CDC_{r,\ell} = FM) \wedge \exists cl, r \in cl \wedge cl \in \bigcup_{\ell'=\ell+1}^{nbLevels} PossiblyEvicted_{\ell',p}$$

Finalement, l'invalidation des lignes de cache provoque un effet indirect sur toutes les références mémoire disposant de la classification de comportement *Always-Miss*. Cette classification est obtenue sans prendre en compte l'invalidation des lignes de cache. Hors, dû au mécanisme d'invalidation, une référence r peut être classifiée *Always-Miss* par l'analyse en étant pourtant présente dans le cache. L'exemple de la figure 2.6 montre que la référence b a sa durée de vie dans le cache prolongée, comparativement à une hiérarchie non-inclusive, dû à l'invalidation de la référence a. Étant donné que l'ensemble des lignes potentiellement évincées est une sur-approximation, déterminer précisément si r produira un succès ou un défaut s'avère impossible. Pour garantir la sûreté, la classification de comportement *Always-Miss* doit donc être modifiée en *Not Classified* pour toutes les références mémoire dans tous les niveaux de cache.

2.4.6 Raffinement de l'analyse

Cette modification de la classification de comportement peut s'avérer trop pessimiste pour le cache de premier niveau. En effet, cette modification ne tient pas compte de la localité spatiale principalement capturée par le premier niveau

de la hiérarchie. À ce niveau, les accès mémoire sont toujours effectués et de ce fait, lors d'un accès à une ligne de cache précédemment invalidée, seul le premier accès à cette ligne, lors d'une succession d'accès, produit un défaut de cache. Pour le premier niveau de cache, la classification de comportement est alors changée en *Not Classified* uniquement si :

$$(CDC_{r,\ell} = AH \vee CDC_{r,\ell} = FM) \wedge \exists cl, r \in cl \wedge cl \in \bigcup_{\ell'=\ell+1}^{nbLevels} PossiblyEvicted_{\ell',p}$$

$$\wedge FirstAccess(r, cl)$$

Avec $FirstAccess(r, cl)$ une fonction booléenne retournant vrai si r est le premier accès à la ligne de cache cl et faux dans le cas contraire.

En se basant sur la propriété d'inclusion, lorsque la taille des lignes de cache d'un niveau est plus grande que celle du niveau précédent, nous pouvons appliquer un autre type de raffinement visant à améliorer la détection de la localité spatiale capturée par les niveaux inférieurs de la hiérarchie. L'idée de ce raffinement est de modifier la classification de comportement en *Always-Hit* des accès successifs à une même ligne de cache dans les niveaux de caches inférieurs disposant de lignes de cache de taille supérieure comparativement à leurs prédécesseurs.

Pour illustrer ce propos, nous considérons deux niveaux de caches successifs ℓ et ℓ' ($\ell < \ell'$) avec des lignes de cache de taille t et respectivement $k * t$ avec $k \in \mathbb{N} \wedge k > 1$. Chacune des lignes de cache du niveau ℓ' est une concaténation de k lignes de cache du niveau ℓ et noté $[cl_1, cl_2, ..., cl_k]$. Suite à un accès à une référence mémoire, nous avons la garantie que cette référence ainsi que la ligne de cache de chaque niveau la contenant est présente dans chacun des niveaux de cache grâce à la propriété d'inclusion. De ce fait, les accès suivants successifs à cette même ligne de cache, pour un niveau de cache donné, sont garantis de produire des succès. Pour le niveau inférieur, ce constat est intéressant car il permet de garantir des succès pour l'ensemble des accès successifs à cette même ligne de cache notamment sur les frontières de lignes de cache du niveau précédent (i.e. premier accès à $cl_2...cl_k$) et ainsi améliorer la détection de la localité spatiale capturée par ce niveau en modifiant la classification de comportement. Plus formellement, la classification de comportement d'une référence mémoire r dans un niveau $\ell > 1$ est modifiée en *Always-Hit* si :

$$\exists cl, r \in cl \wedge \neg FirstAccess(r, cl) \wedge LineSize(\ell) > LineSize(\ell - 1)$$

2.4.7 Terminaison de l'analyse

L'analyse statique des hiérarchies de caches inclusives repose sur les mêmes calculs de point fixe que l'analyse statique des hiérarchies de caches non-inclusives. De ce fait, la preuve de terminaison du calcul de point fixe est une transposition directe de la preuve de terminaison précédente.

2.4.8 Exemple

Nous appliquons notre analyse sur l'exemple précédent en regardant l'impact des invalidations sur le contenu du cache de premier niveau. La figure 2.8 présente la chaîne d'accès ainsi que les états abstraits de cache de l'analyse *Must* du premier niveau (L1) et de l'analyse *Persistence* du second niveau (L2) au point de programme $p1$ et $p2$. Nous avons précédemment observé (figure 2.6) que lors de l'accès à la référence c entre les points $p1$ et $p2$, la référence a est évincée du cache de second niveau et de ce fait invalidée dans le cache de premier niveau. Au niveau de notre analyse, la référence a est présente dans l'état abstrait de cache du premier niveau de l'analyse *Must* au point $p2$ ce qui amène à la classification de comportement *Always-Hit* lors de l'accès effectué au point $p3$ lors de l'analyse de cache. Cependant la référence a est également présente dans la voie virtuelle de l'analyse de *Persistence* du second niveau au point $p2$ conduisant le post traitement, en charge de la prise en compte du mécanisme d'invalidation, à changer la classification de comportement de la référence a en *Not Classified* pour l'accès au cache de premier niveau réalisé au point $p3$.

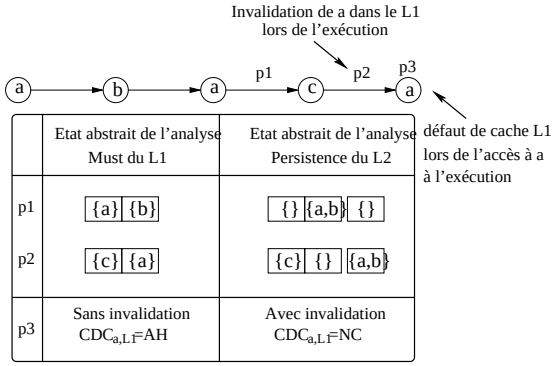


FIG. 2.8: Impact de la prise en compte des invalidations sur la classification de comportement. La référence a au point de programme $p3$ est classifiée *Always-Hit* lors de l'analyse mais comme celle-ci est également présente dans la voie virtuelle de l'analyse *Persistence* du niveau inférieur, sa classification est modifiée en *Not Classified* suite à la prise en compte des invalidations.

2.5 Analyse statique des hiérarchies de caches exclusives

2.5.1 Fonctionnement et propriétés

Les hiérarchies de caches exclusives assurent la propriété d'exclusion entre les différents niveaux de caches en garantissant que chacune des lignes de cache présente dans la hiérarchie est stockée une fois et une seule dans l'ensemble des niveaux de cache composant la hiérarchie. Cette stratégie d'exclusion entre les niveaux permet d'accroître virtuellement la taille de la hiérarchie de cache en évitant la redondance d'information [112].

La solution employée pour maintenir la propriété d'exclusion entre les différents niveaux lors d'un défaut de cache dans le premier niveau consiste à charger la ligne de cache, contenant la référence mémoire, uniquement dans ce niveau et à invalider cette ligne des niveaux inférieurs si elle était présente avant l'accès. Chacun des niveaux suivants stockent les lignes de cache uniquement lorsqu'elles sont évincées du niveau précédent.

Plus formellement, nous introduisons les propriétés suivantes définissant les hiérarchies de cache exclusives considérées :

P1. Une information est recherchée dans le cache de niveau ℓ si et seulement si un défaut de cache se produit dans le cache de niveau $\ell - 1$, sauf pour le cache de premier niveau qui est toujours accédé ;

P2. L'âge des lignes de cache contenues dans un niveau ℓ est mis à jour, en fonction de la politique de remplacement de cache, à chaque fois qu'une ligne de cache est insérée dans le cache de niveau ℓ ;

P3 L'âge des lignes de cache contenues dans le premier niveau de cache est mis à jour, en fonction de la politique de remplacement de cache, lors de chaque chaque accès à ce niveau ;

P4. A chaque fois qu'un défaut de cache se produit dans le cache de premier niveau, la ligne de cache contenant l'information provoquant ce défaut est intégralement insérée dans le cache de premier niveau ;

P5. Chaque ligne de cache est contenue au plus une fois dans la hiérarchie de cache. Cette exclusion est assurée de la façon suivante :

- P5a. A chaque fois qu'une ligne de cache est insérée dans le cache de premier niveau, cette ligne est invalidée du niveau inférieur dans lequel elle était éventuellement présente ;
- P5b. Une ligne de cache cl est insérée dans le niveau $\ell > 1$ suite à son éviction du niveau $\ell - 1$.

P6. La taille des lignes de cache de chacun des niveaux de la hiérarchie est identique ;

P7. Il n'y a aucune autre action sur le contenu des caches (i.e. recherches,

modifications, invalidation...) autres que celles mentionnées ci-dessus.

La propriété P1, est la seule de toutes les propriétés à être commune entre les différentes hiérarchies de caches en assurant une recherche en séquence des références mémoire dans l'ensemble de la hiérarchie.

La propriété P6, indiquant que la taille des lignes de cache est identique pour les différents niveaux, est couramment utilisé pour les hiérarchies de caches exclusives afin de simplifier le transfert des lignes de cache entre les différents niveaux.

Les propriétés P2 à P5 définissent les différents traitements réalisés lors d'un accès mémoire. Dans le cas d'un succès dans le cache de premier niveau, le traitement est similaire aux autres hiérarchies, l'information est retournée au processeur et l'âge des lignes de cache est mis à jour en fonction de la politique de remplacement (propriété P3). Par contre, dans le cas d'un défaut de cache dans le premier niveau, si la ligne de cache cl, contenant l'information manquante, est présente dans un cache de niveau $\ell > 1$, cl est invalidée au niveau ℓ (propriété P5a) après avoir été insérée dans le cache de premier niveau (propriété P4). Le chargement de cl dans le cache de premier niveau peut provoquer l'éviction d'une ligne de cache cl' à ce niveau. Dans ce cas, cl' est alors stockée dans le niveau suivant (propriété P5b). En appliquant la politique de remplacement (propriété P2), le second niveau peut à son tour évincer une ligne de cache cl'' provoquant son chargement dans le niveau suivant et ainsi de suite jusqu'au dernier niveau de cache.

La figure 2.9 illustre un tel traitement, pour une hiérarchie de caches constituée de deux niveaux, lors d'un accès à une référence $b5$ contenue dans le cache de second niveau. $b5$ est alors chargée dans le premier niveau de cache ce qui évince la référence $b2$. $b5$ est également invalidée du second niveau ce qui laisse un bloc de cache libre pour stocker la référence $b2$ à sa place. L'âge des différentes lignes de cache, indiqué au dessus de chaque état de cache concret, est mis à jour lors de ce traitement.

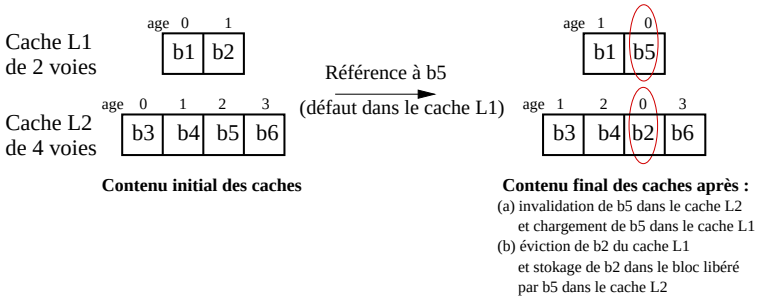


FIG. 2.9: Exemple du traitement effectué par une hiérarchie de caches exclusive lors d'un l'accès à la référence $b5$.

Afin de réaliser la permutation du bloc $b2$ et $b5$, entre les deux niveaux de caches, illustrée par la figure 2.10, nous supposons que l'architecture dispose de mémoires tampons d'évincement entre chaque couple de niveaux consécutifs pour stocker temporairement les lignes de cache évincées (à droite) ainsi qu'un tampon de chargement servant à stocker temporairement la ligne de cache contenant l'information recherchée (à gauche). Avec ces mémoires tampons, le traitement d'un accès peut être réalisé en deux phases successives. La première consiste à rechercher l'information successivement dans chacun des niveaux tant que la ligne de cache n'est pas trouvée. Pour un niveau donné et si la ligne de cache est absente, la ligne de cache qui sera évincée en appliquant la politique de remplacement est d'ores et déjà connue et peut donc être stockée dans la mémoire tampon d'évincement. Si par contre, la ligne de cache est trouvée, elle peut être transférée dans la mémoire tampon de chargement. La seconde phase, finalisant le traitement d'un accès, charge l'information recherchée dans le cache de premier niveau et transfère les lignes de cache évincées dans les niveaux inférieurs en prenant les blocs de cache libérés lors de la première phase. Si jamais l'information recherchée n'est pas présente dans la hiérarchie de caches, l'information est chargée directement à partir de la mémoire principale et les lignes de cache évincées sont gérées de façon identique.

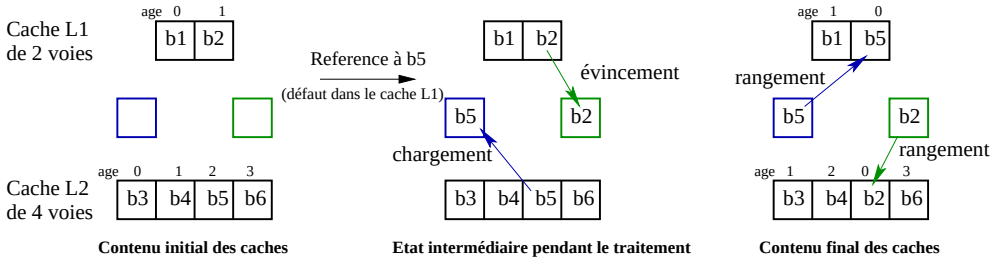


FIG. 2.10: Détail du traitement par une hiérarchie de caches exclusive lors d'un l'accès à la référence $b5$. L'accès est effectué en deux temps : (1) éviction et chargement dans des mémoires tampons des références concernées, (2) chargement des références présentes dans les tampons dans les caches.

2.5.2 Problématique de l'analyse

Le fait de charger une ligne de cache dans un niveau donné exclusivement lors de son éviction, par le niveau précédent, pour les caches autres que le premier niveau, rend la date de l'accès à une référence mémoire indépendante de celle de son chargement dans les niveaux inférieurs de la hiérarchie. Cette indépendance nécessite une approche différente, par rapport aux analyses proposées précédemment, pour analyser les hiérarchies de caches exclusives. En effet, l'analyse des

hiérarchies précédentes repose sur la notion de classification d'accès au cache modélisant les flux d'accès entre les niveaux successifs. Cette classification permet de déterminer, lors de l'analyse, si un accès à une référence mémoire est propagé à un niveau donné et si, de ce fait, la ligne de cache contenant cette référence est à charger à cet instant dans ce niveau. Pour les hiérarchies de caches exclusives, l'instant de chargement étant décorrelé de celui de l'accès, l'utilisation de la classification d'accès au cache n'est donc pas appropriée pour les analyser.

2.5.3 Modélisation et analyse des hiérarchies de caches exclusives

L'analyse que nous proposons pour les hiérarchies de caches exclusives repose sur une modélisation de la hiérarchie basée sur les propriétés de l'exclusion. Jusqu'ici, la notion d'âge associée à une ligne de cache était locale à chaque niveau de cache composant la hiérarchie. Cependant, grâce, d'une part, à l'unicité des lignes de cache au sein des différents niveaux composant la hiérarchie (propriété P5) et, d'autre part, à la façon de stocker les lignes de cache dans les niveaux inférieurs lors de leurs évictions (propriété P5b), nous pouvons étendre la notion d'âge d'une ligne de cache à tous les niveaux de la hiérarchie et ainsi obtenir un âge global pour chacune des lignes de cache. L'âge global d'une ligne de cache cl présente dans un niveau de cache ℓ s'obtient en additionnant les degrés d'associativités des niveaux de caches précédant ℓ avec l'âge local de cl dans le niveau ℓ. Par exemple, sur la figure 2.9, l'âge de la référence $b5$ avant de réaliser l'accès est de deux localement dans le cache de second niveau. En ajoutant le degré d'associativité du premier niveau, nous obtenons l'âge global de la référence $b5$, soit quatre dans cet exemple.

Cette notion d'âge global nous permet de modéliser l'ensemble des niveaux de cache composant la hiérarchie par un unique état, que nous appelons *état abstrait de la hiérarchie*, en concaténant chacun des niveaux, en commençant par le premier. La figure 2.11 montre l'état abstrait de la hiérarchie obtenu à partir de l'exemple de la figure 2.9 avant l'accès à la référence $b5$ pour le cas particulier où le nombre d'ensembles du L1 est égal à celui du L2.

Cette représentation sous la forme d'un état abstrait de la hiérarchie se rapproche des états abstraits modélisant les caches dans le sens où, chacun des éléments est assimilable à un bloc de cache stockant une ligne de cache lorsque celle-ci est évincée du bloc précédent. Une telle représentation permet de ramener l'analyse des hiérarchies de caches exclusives à l'analyse d'un unique niveau de cache en se basant sur les approches existantes comme [101] (détaillée précédemment dans le chapitre 1, section 1.4.4) que nous utilisons. Cette modélisation permet à l'analyse de se focaliser uniquement sur l'analyse des contenus des différents niveaux de la hiérarchie. En effet, la modélisation des flux d'accès entre

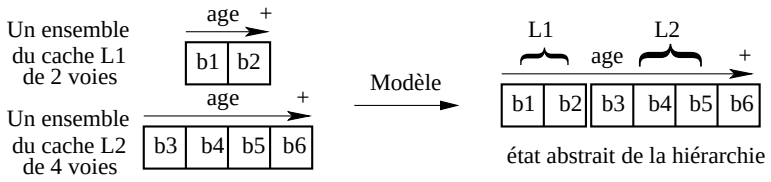


FIG. 2.11: Modélisation des hiérarchies de caches exclusives sous la forme d'un état abstrait de la hiérarchie (cas particulier où le nombre d'ensembles du L1 est égal à celui du L2).

niveaux est directement effectuée par l'analyse de cache utilisée. Ils nous suffit donc de voir comment appliquer les fonctions *Join* et *Update* sur un état abstrait de la hiérarchie pour obtenir une analyse applicable aux hiérarchies de caches exclusives. Nous définissons les fonctions *Join* et *Update* à partir des fonctions originales définies pour l'analyse de cache mono-niveau.

Définition de la fonction *Join*. La fonction *Join* originale des analyses *Must*, *May* et *Persistence* est directement applicable sur l'état abstrait de la hiérarchie et revient à réaliser, soit l'union (*May* et *Persistence*), soit l'intersection (*Must*), de l'ensemble des lignes de cache présentes dans la hiérarchie en conservant, soit l'âge minimal (*May*), soit l'âge maximal (*Must* et *Persistence*), de chacune des lignes de cache pour construire l'état abstrait de la hiérarchie résultant.

Définition de la fonction *Update*. À la différence des états de cache abstraits où les lignes de cache évincées d'un bloc sont insérées dans le bloc suivant du *même* ensemble de cache, les lignes de cache évincées d'un niveau de cache de l'état abstrait de la hiérarchie ne sont pas nécessairement insérées dans un *unique* ensemble du niveau suivant. En effet, suivant la mise en œuvre de la hiérarchie, trois configurations, illustrées par la figure 2.12, peuvent être observées concernant la projection des lignes de cache entre les ensembles de deux niveaux consécutifs :

1. L'ensemble des lignes de cache pouvant être projeté dans un ensemble du cache de niveau ℓ est également projeté dans un seul ensemble du cache de niveau $\ell + 1$ (figure 2.12.a). Ce cas de figure se produit dès lors que les nombres d'ensembles des niveaux ℓ et $\ell + 1$ sont égaux.
2. L'ensemble des lignes de cache pouvant être projeté dans un ensemble du cache de niveau ℓ est projeté dans plusieurs ensembles du cache de niveau $\ell + 1$ (figure 2.12.b). Ce cas de figure se produit dès lors que le nombre d'ensembles du niveau ℓ est inférieur à celui du niveau $\ell + 1$.
3. L'ensemble des lignes de cache pouvant être projeté dans plusieurs ensembles du cache de niveau ℓ est projeté dans un seul ensemble du cache de niveau $\ell + 1$ (figure 2.12.c). Ce cas de figure se produit dès lors que le

nombre d'ensembles du niveau ℓ est supérieur à celui du niveau $\ell+1$. C'est le cas des caches de victimes [46] servant à réduire le temps d'accès résultant des défauts de cache dus aux conflits.

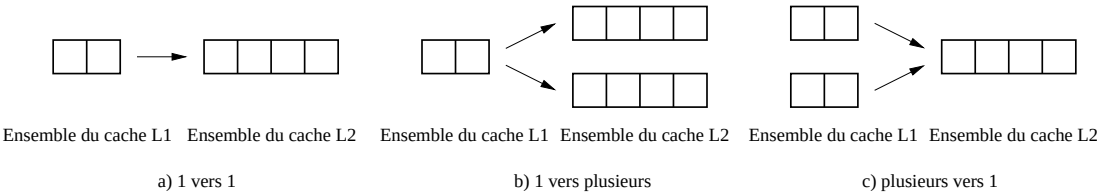


FIG. 2.12: Illustration des trois relations entre ensembles possibles entre deux niveaux de cache disposant d'une politique de gestion exclusive.

Afin d'appliquer la fonction *Update* sur l'état abstrait de la hiérarchie, il nous faut donc préalablement déterminer le ou les ensembles du niveau $\ell+1$ susceptibles de recevoir des lignes de cache évincées par le niveau ℓ. Pour ce faire, nous introduisons la fonction *next_sets* qui prend en paramètre le contenu évincé des blocs de cache du niveau ℓ et qui retourne les sous-ensembles des lignes de cache, éventuellement vide, à insérer dans chacun des ensembles du niveau $\ell+1$ pouvant recevoir ces lignes.

Pour illustrer le fonctionnement de la fonction *next_sets*, revenons sur les différentes relations entre ensembles possibles des hiérarchies de caches exclusives (figure 2.12). Pour le cas où les lignes de cache évincées ne peuvent être qu'uniquement projetées dans un seul ensemble de cache du niveau suivant (figure 2.12.a et 2.12.c), la fonction *next_sets* retourne l'ensemble des lignes de cache évincées à insérer dans l'unique ensemble de destination du niveau suivant. Pour le cas où les lignes de cache évincées peuvent être projetées dans différents ensembles de cache du niveau suivant (figure 2.12.b), la fonction *next_sets* trie chacune de ces lignes de cache en les regroupant dans des ensembles en fonction de l'ensemble de cache où elles sont projetées dans le niveau suivant. Ainsi, un ensemble de lignes de cache à insérer est déterminé pour chaque ensemble possible de destination. Dans l'exemple de la figure 2.12.b, nous obtenons avec cette fonction deux ensembles contenant les lignes de cache évincées du cache L1, un pour chaque ensemble de cache de destination du cache L2.

La fonction *next_sets* est une formalisation explicite d'une fonction existante au sein de la fonction *Update* originale. En effet, chaque ligne de cache évincée d'un bloc de cache est implicitement assignée au bloc suivant de l'ensemble du cache, pour modéliser la mise à jour de son âge. La fonction *Update* pour les hiérarchies de caches exclusives fonctionne donc de façon similaire à la fonction *Update* originale, à l'exception près que la fonction *next_sets* est utilisée lors de

la mise à jour de l'âge des lignes de cache entre deux niveaux de cache consécutifs de la hiérarchie.

L'utilisation de la fonction *next_sets* entre deux niveaux de la hiérarchie peut déterminer plusieurs ensembles où des lignes de cache seront à insérer dans le niveau $\ell + 1$. Cette particularité a pour effet de mettre à jour l'âge de toutes des lignes de cache contenues dans les ensembles ainsi déterminés du niveau $\ell+1$, afin de garantir la sûreté des analyses maintenant un âge maximal (*Must* et *Persistence*).

Par contre, l'utilisation directe de la fonction *next_sets* au sein de la fonction *Update* peut, dans certains cas, mettre en défaut la sûreté de l'analyse *May*. En effet, le fait de mettre à jour l'âge des lignes de cache contenues dans les ensembles du niveau $\ell + 1$ susceptibles de recevoir des lignes de cache évincées peut incrémenter l'âge de lignes de cache non mises à jour lors de l'exécution. Ce phénomène peut se produire dès lors que plusieurs ensembles de cache du niveau $\ell + 1$ sont susceptibles de recevoir des lignes de cache évincées, et que la fonction *next_sets* détermine au moins un ensemble de lignes de cache vide à insérer. Pour contourner ce phénomène, il suffit, lors de l'analyse *May*, de modifier uniquement l'âge des lignes de cache contenues dans les ensembles de cache devant recevoir au moins une ligne de cache (i.e. les ensembles de lignes de cache non vides déterminés par la fonction *next_sets*).

Déduction de la classification de comportement. La classification de comportement de chaque accès dans chacun des niveaux de la hiérarchie est déterminée en fonction de l'âge de la ligne de cache contenant la référence mémoire au sein de l'état abstrait de la hiérarchie résultant de chaque analyse. Pour les analyses conservant l'âge maximal (*Must* et *Persistence*), la référence est présente dans un niveau donné si son âge est compris entre la première et la dernière voie incluse de ce niveau, et dans ce cas, elle est classifiée *Always-Hit* pour l'analyse *Must*, et *First-Miss* pour l'analyse de *Persistence*. Sinon, pour l'analyse *May*, conservant l'âge minimal, la référence est considérée comme potentiellement présente et classifiée *Not Classified* dans un niveau donné si l'âge de la ligne de cache est inférieur ou égal à la dernière voie de ce niveau et *Always-Miss* dans le cas contraire.

Bénéfices de la modélisation Les bénéfices de cette modélisation, sous la forme d'un état abstrait de la hiérarchie, sont que les analyses de cache mono-niveau existantes peuvent être directement utilisées pour analyser les hiérarchies de caches exclusives. En particulier, la sûreté et la terminaison des l'analyses sont des résultats directs de la sûreté et de la terminaison de ces analyses. Finalement, cette modélisation se focalise uniquement sur le contenu de la hiérarchie car elle nous permet de s'abstraire du mécanisme d'invalidation forçant l'exclusion, et

donc d'une source de pessimisme, grâce à l'utilisation des fonction *Update* et *Join* garantissant l'unicité des lignes de cache au sein de l'état abstrait de la hiérarchie.

2.5.4 Exemple

La figure 2.13 illustre le fonctionnement de la fonction *Update* lors d'un accès à une référence mémoire b lors des analyses *Must* et *May* sur une hiérarchie de caches exclusives constituée de trois niveaux. Par souci de concision, seule la tranche de la hiérarchie affectée par l'accès à la référence b est représentée. L'ensemble des références mémoire est projeté dans un unique ensemble de cache dans le premier (L1) et le dernier (L3) niveau tandis que les références paires, d'après l'ordre lexicographique, sont projetées dans l'ensemble du bas dans le second niveau (L2) et les références impaires dans celui du haut.

Lors de l'accès à la référence b, celle-ci étant présente dans la hiérarchie, elle est invalidée du L3 et insérée dans le L1, ce qui évince les références a et c. La fonction *next_set* détermine que les références a et c sont à insérer dans l'ensemble du haut du L2 tandis qu'aucune référence évincée n'est à insérer dans l'ensemble du bas. Au niveau de l'analyse *Must*, les âges des références contenues dans les deux ensembles sont mis à jour ce qui évince les références d (ensemble du bas) et e (ensemble du haut) au niveau du L2 et elles sont de ce fait insérées dans le L3. Concernant l'analyse *May*, uniquement l'ensemble du haut du L2 est mis à jour car aucune référence évincée n'est à insérer dans l'ensemble du bas. La référence d est ainsi conservée dans le L2 tandis que la référence e est évincée du L2 et insérée dans le L3.

La classification de comportement déterminée par l'analyse pour l'accès à la référence b est *Not Classified* pour le cache L1 et le cache L2 car la référence b, avant l'accès, est présente dans le troisième niveau de l'état abstrait de la hiérarchie et donc de fait classifié *Always-Hit* pour le cache L3.

2.6 Extension à d'autres politiques de remplacement de cache

Pour le moment, nous avons considéré des hiérarchies de caches où la politique de remplacement de cache LRU est mise en œuvre pour chacun des niveaux. Dans cette section, nous allons étendre nos analyses afin de supporter différentes politiques de remplacement de cache non-LRU.

La méthode d'analyse statique mono-niveau [101] a été en partie étendue pour prendre en compte les politiques de remplacement *Pseudo-LRU* et *Pseudo-Round-Robin* dans [43]. Cependant, seulement les analyses *Must* et *May* ont été

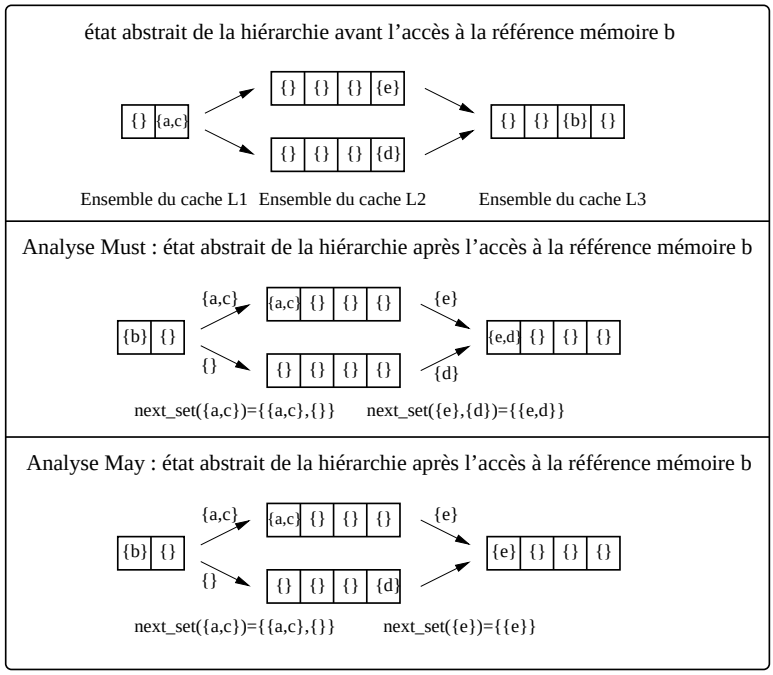


FIG. 2.13: Utilisation de la fonction *Update* sur l'état abstrait de la hiérarchie.

définies pour ces politiques de remplacement ce qui produit une estimation du pire temps d'exécution plus pessimiste comparativement à une analyse disposant également de l'analyse *Persistence*.

Nous proposons de repartir de l'analyse mono-niveau [101], plus précisément avec la version modifiée (fonction $Update_m$, paragraphe 2.3.3) et de l'étendre en modifiant la représentation des états de cache abstraits afin de prendre en compte les politiques de remplacement de cache : *Pseudo-LRU*, *MRU*, *FIFO* et *RANDOM*.

Pour ce faire, nous nous basons sur les résultats théoriques [86] définissant deux bornes nommées respectivement *minimum life-span (mls)* et *evict* pour chacune des politiques de remplacement mentionnées précédemment. La borne mls détermine la durée de vie minimale d'une ligne de cache dans un ensemble de cache de degré d'associativité k. Par opposition, la borne *evict* détermine la durée de vie maximale d'une ligne de cache dans un ensemble de cache de degré d'associativité k. En d'autres termes, ces deux bornes déterminent le nombre minimal/maximal d'accès distincts, dans un ensemble de cache, différents de la ligne de cache cl pour évincer cl.

Cependant, dans [86] les bornes mls et *evict* sont définies en considérant une séquence d'accès à des éléments tous distincts. Pour être utilisable en pratique, nous devons redéfinir ces bornes pour des séquences de taille arbitraires constituées de n accès distincts. Le tableau 2.2 présente les bornes mls et *evict* modifiées pour traiter les séquences d'accès quelconques constituées de n accès distincts pour l'ensemble des politiques de remplacement de cache considérées. La preuve de chacune des bornes mls et *evict* est fournie en annexe de ce document.

	LRU	Pseudo-LRU	MRU	FIFO	RANDOM
mls	k	$log_2(k)+1$	2	1	1
evict	k	2 si $k=2$ et ∞ sinon	$2k-2$	$2k-1$	∞

TAB. 2.2: Les bornes mls et *evict* modifiées pour $k \geq 2$

2.6.1 Utilisation des bornes mls et *evict*

Nous utilisons la borne mls pour réduire le nombre de voies de l'état de cache abstrait des analyses garantissant la présence dans le cache d'une référence (i.e. les analyses *Must* et *Persistence*). De façon similaire, nous utilisons la borne *evict* pour augmenter le nombre de voies de l'état de cache abstrait de l'analyse *May* déterminant si une référence peut être présente dans le cache. Le nombre de voies des états de cache abstraits devient de ce fait décorrélé du degré d'associativité du cache. La figure 2.14 illustre le traitement réalisé sur les états de cache abstraits pour la politique de remplacement MRU.

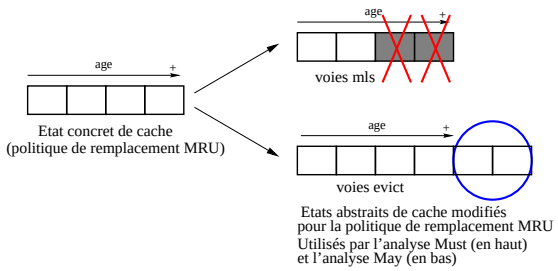


FIG. 2.14: Utilisation des bornes *mls*/*evict* sur les états de cache abstraits afin de modéliser la politique de remplacement de cache MRU.

La modification, portant uniquement sur la représentation des états de cache abstraits, permet d'être indépendant du type de hiérarchie de cache analysée et les analyses *Must*, *May* et *Persistence* peuvent être directement appliquées. Néanmoins, pour l'analyse des hiérarchies de caches exclusives, cette modification doit être réalisée sur chaque niveau de cache avant la construction de la séquence ordonnée.

2.6.2 Sûreté et terminaison de l'analyse

La sûreté des analyses utilisant les bornes *mls* et *evict* repose d'une part sur les preuves de ces bornes pour chacune des politiques de remplacement de cache et d'autre part sur l'analyse prouvée sûre pour une taille d'état abstrait de cache donnée. La preuve de terminaison est, quant à elle, une transposition directe des preuves de terminaison précédentes.

2.7 Calcul de l'estimation du pire temps d'exécution

Le résultat de l'analyse statique de chacune des différentes hiérarchies de caches donne le comportement pire cas de chaque accès dans chacun des niveaux avec la classification de comportement. Dans ce paragraphe, nous présentons les formules permettant de calculer la contribution de chaque accès mémoire au pire temps d'exécution en considérant la hiérarchie de cache analysée dans son ensemble.

La caractéristique commune entre les différentes politiques de gestion est que lors d'un accès, la recherche de l'information s'effectue en parcourant de façon séquentielle chaque niveau de la hiérarchie en commençant par le niveau le plus haut. De ce fait, Pour chaque accès à une référence mémoire r, sa contribution

au pire temps d'exécution peut s'exprimer comme la somme des latences d'accès de chacun des niveaux de la hiérarchie mémoire où l'accès est propagé. Ce calcul, détaillé ci-dessous, s'effectue en parcourant la hiérarchie mémoire à partir du cache de premier niveau et jusqu'à se que l'on trouve une classification de comportement exprimant un succès (i.e. *Always-Hit* ou *First-Miss*) dans un des niveaux de cache, ou si l'accès est propagé à la mémoire principale. Lors de ce calcul, la classification de comportement *Not Classified* est considérée comme produisant un défaut de cache. Cette supposition est sûre pour les architectures sans anomalie temporelle [61, 104, 87]. Pour les architectures pouvant générer des anomalies temporelles (i.e. les architectures avec des pipelines à exécution dans le désordre), une analyse plus fine doit être utilisée, comme celle proposée dans [51], explorant l'intervalle de temps d'exécution des instructions.

La présence de la classification *First-Miss* nécessite une distinction entre le premier accès et les suivants à la référence mémoire r. Pour ce faire, nous introduisons deux variables binaires intermédiaires $first\_access\_may\_occur_\ell(r)$ et $next\_access\_may\_occur_\ell(r)$. Ces variables représentent le fait que l'accès à la référence r peut se produire (1) ou non (0) au niveau ℓ respectivement lors du premier accès et lors des accès suivants. Elles sont mises à 0 lorsque qu'une classification de comportement exprimant un succès est détectée dans un niveau précédent.

$$first\_access\_may\_occur_\ell(r) = \begin{cases} 1 \text{ si} & \ell = 1 \\ 1 \text{ si} & first\_access\_may\_occur_{\ell-1}(r) = 1 \\ & \wedge \ (CDC_{r,\ell-1} = AM \\ & \quad \vee CDC_{r,\ell-1} = FM \\ & \quad \vee CDC_{r,\ell-1} = NC) \\ 0 & \text{sinon} \end{cases}$$

$$next\_access\_may\_occur_\ell(r) = \begin{cases} 1 \text{ si} & \ell = 1 \\ 1 \text{ si} & next\_access\_may\_occur_{\ell-1}(r) = 1 \\ & \wedge \ (CDC_{r,\ell-1} = AM \\ & \quad \vee CDC_{r,\ell-1} = NC) \\ 0 & \text{sinon} \end{cases}$$

Les variables $first\_access\_may\_occur_\ell(r)$ et $next\_access\_may\_occur_\ell(r)$ peuvent ensuite être utilisées pour calculer la contribution des accès mémoire au pire temps d'exécution de la référence r lors du premier accès ainsi que les suivants, représentée par les variables $COST\_first(r)$ et $COST\_next(r)$ et calculée de la façon suivante :

$$COST\_first(r) = \sum_{\ell=1}^{nbLevels+1} Latency_\ell * first\_access\_may\_occur_\ell(r)$$

$$COST\_next(r) = \sum_{\ell=1}^{nbLevels+1} Latency_\ell * next\_access\_may\_occur_\ell(r)$$

Avec $Latency_\ell$ la constante correspondant à la latence d'accès au cache de niveau ℓ et $nbLevels+1$ représentant la mémoire principale, située après le dernier niveau de cache de la hiérarchie, contenant l'intégralité de la tâche analysée.

Les variables $COST\_first(r)$ et $COST\_next(r)$ fournissent ainsi la contribution des accès mémoire au pire temps d'exécution de la référence r en un point de programme donné. Cette information peut ensuite être intégrée dans les méthodes existantes d'estimation du pire temps d'exécution comme la méthode IPET (en anglais : *Implicit Path Enumeration Techniques*) [81] ou encore la méthode à base d'arbre syntaxique [80].

2.8 Expérimentations

2.8.1 Conditions expérimentales

Programme de test. Les expérimentations sont menées sur dix programmes de test et deux tâches d'une application réelle. Les programmes de test utilisés sont les *WCET benchmarks* maintenus par le groupe de recherche en WCET de Mälardalen [62]. Les tâches réelles sont issus d'une application réelle nommée debie [28] fournie par le *Space Systems Finland Ltd (SSF)* s'exécutant sur satellite afin de mesurer l'impact des micro-météorites et petit débris. Le tableau 2.3 donne les caractéristiques de chacun des programmes de test ainsi que des tâches de debie utilisées.

Analyse de caches et estimation du pire temps d'exécution. Les expérimentations sont menées sur du code binaire compilé avec gcc 4.1 sans optimisation pour du MIPS R2000/R3000 [66]. L'estimation du pire temps d'exécution est calculé par le logiciel HEPTANE [24] en utilisant la méthode d'énumération implicite des chemins (IPET). Les analyses *Must*, *May* et *Persistence* sont appliquées sur chaque niveau de la hiérarchie de cache. L'analyse est contextuelle et de fait l'analyse est effectuée pour chaque fonction sur chacun des contextes d'appels.

Nom	**Description**	**Taille du code** (octets)
matmult	Multiplication de matrices 50x50 composées de nombre entiers	1200
ns	Recherche dans un tableau multi-dimensionel	600
bs	Recherche binaire dans un tableau de 15 entiers	336
minver	Inversion d'une matrice 3x3 composée de nombre flottants	4408
jfdctint	Transformée en cosinus discrète	3040
crc	Contrôle de redondance cyclique	1432
qurt	Calcul des racines d'équations quadratiques	1928
fft	Transformée de Fourier rapide	3536
adpcm	Encodage de la voix	7740
statemate	Code généré automatiquement par STARC (STAtechart Real-time-Code generator)	8900
health monitoring	Vérification périodique de l'état du système, incluant la récupération de valeurs des capteurs.	14944
telecommand	Traitement et répartition vers les composants concernés des commandes reçues à distance	14824

TAB. 2.3: Caractéristiques des programmes de test

Pour séparer l'effet des caches des autres mécanismes matériels, l'estimation du pire temps d'exécution considère uniquement la contribution des accès mémoire au pire temps d'exécution comme présentée dans la section 2.7. Les effets des autres mécanismes matériels ne sont pas pris en compte. En particulier, nous ne prenons pas en compte les anomalies temporelles causées par les interactions entre les caches et les pipelines. La classification de comportement *Not Classified* peut alors être considérée comme ayant le même comportement pire cas que la classification *Always-Miss* pendant le calcul de l'estimation du pire temps d'exécution. Finalement, l'analyse débute en considérant un état de cache vide ce qui est sûr en absence d'anomalies temporelles en utilisant une politique de remplacement de cache LRU et pour les politiques de remplacement non-LRU en utilisant les bornes $mls/evict$ définies indépendamment du contenu initial du cache.

Environnement de mesures. Les mesures effectuées dans le pire scénario d'exécution utilisent un simulateur de jeu d'instructions MIPS [70]. En raison de la difficulté à identifier les données d'entrées conduisant à la pire situation pour les programmes complexes, nous comparons l'estimation du pire temps d'exécution avec les valeurs mesurées uniquement pour les programmes de test simples (*matmult, ns, bs, minver, jfdctint*). Tous ces programmes de test sont à chemin unique sauf *bs* qui est suffisamment simple pour identifier ces données d'entrées conduisant au pire scénario d'exécution.

Le simulateur a été étendu pour supporter une hiérarchie de cache constituée de trois niveaux avec les différentes politiques de gestion : *non-inclusive*, *inclusive* et *exclusive*. En raison des effets dominos [13] pouvant se produire avec les politiques de remplacement de cache non-LRU, l'estimation du pire temps d'exécution est comparée uniquement avec les mesures effectuées avec la politique de remplacement de cache LRU. Pour les autres politiques de remplacement de cache, trouver le pire temps d'exécution même dans le cas de programme à chemin unique requière une exploration exhaustive de tous les états initiaux de cache ce qui est évidemment trop coûteux.

Configuration de cache. Les expérimentations sont menées en utilisant une hiérarchie de caches composée de deux niveaux avec différentes configurations de cache résumées dans le tableau 2.4. Nous utilisons deux groupes de configurations avec différentes tailles de cache (*small-32-32* et *small-32-64* pour les programmes de test ; *medium-32-32* et *medium-32-64* pour debie). Pour chaque groupe, les deux configurations sont différenciées par la taille des lignes de cache du second niveau (L2). Les configurations avec les mêmes tailles de lignes de cache dans le cache de premier (L1) et de second niveau (L2) sont utilisées pour détecter si l'analyse statique de cache capture la localité temporelle au niveau du L2. Les configurations avec des tailles de lignes différentes sont quant à elles utilisées pour détecter si l'analyse statique de cache capture les deux types de localité, spatiale et temporelle, au niveau du L2.

	Taille du cache		**Taille des lignes de cache**		**Associativité**	
	L1	L2	L1	L2	L1	L2
small-32-32	1KB	2KB	32B	32B	4	8
small-32-64	1KB	2KB	32B	64B	4	8
medium-32-32	8KB	16KB	32B	32B	4	8
medium-32-64	8KB	16KB	32B	64B	4	8

Tab. 2.4: Les différentes configurations de cache

Métriques. Afin d'évaluer la précision de notre approche, la comparaison du taux de succès dans le cache L2 entre l'analyse statique et les mesures n'est pas appropriée en raison du pessimisme inhérent à l'analyse statique du cache L1, introduisant des accès au niveau du cache L2 lors de l'analyse ne se produisant en pas nécessairement lors de l'exécution. À la place, nous utilisons les trois métriques suivantes :

a. *Le nombre d'accès et le nombre de défauts de cache se produisant dans chaque niveau de la hiérarchie.* Ces valeurs sont fournies pour l'analyse

statique et pour les valeurs obtenues par mesure lors de l'exécution de la tâche dans le pire scénario d'exécution ;

b. *La contribution au pire temps d'exécution des accès mémoire.* Afin de montrer l'intérêt d'une analyse multi-niveaux, deux valeurs sont données : une en considérant la hiérarchie de cache (L1+L2) et l'autre en ignorant le cache L2 (dans ce cas tous les accès au L2 sont considérés comme produisant des défauts de cache). Des latences de 1 cycle pour le L1, de 10 cycles pour le L2 et de 100 cycles pour la mémoire principale sont utilisées. Lorsque le L2 est ignoré lors de l'analyse, la latence mémoire considérée est de 110 cycles ;

c. *L'amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution.* Cette valeur est calculée pour une configuration de cache analysée comparée à une configuration de cache de référence. La configuration de référence est toujours celle disposant d'une politique de gestion *non-inclusive* (NI) avec la même politique de remplacement de cache que la configuration analysée ($\frac{Analysis_{evaluated}}{Analysis_{NI}} - 1$). Plus cette valeur est petite, plus le résultat de l'analyse de cache de la configuration analysée est meilleur comparativement à la configuration de référence.

2.8.2 Résultats expérimentaux

2.8.2.1 Résultats pour la politique de gestion *non-inclusive* disposant de la politique de remplacement de cache LRU

Afin d'évaluer la précision de l'analyse de cache multi-niveaux, les résultats de l'analyse statique sont comparés avec ceux obtenus en exécutant les programmes dans leur scénario pire cas. La précision de l'analyse est évaluée sur les programmes de test simples (*matmult, ns, bs, minver, jfdctint*) pour lesquels déterminer le scénario pire cas s'avère relativement simple. Les configurations de cache utilisées sont *small-32-32* et *small-32-64* afin que le code de la plupart des programmes de test ne tienne pas dans le cache L2. Pour chaque configuration, les résultats sont fournis pour la valeur estimée par l'analyse statique (colonne de gauche) et pour la valeur mesurée (colonne de droite) dans le tableau 2.5 avec les métriques *a* et *b*. Dans le cas des mesures, une seule valeur est donnée dans la colonne de droite pour la contribution des accès mémoire au pire temps d'exécution et elle représente la valeur obtenue en considérant une architecture avec un cache L1 et un cache L2.

Deux types de comportements peuvent être observés en fonction de la structure de l'application, qui impacte fortement la précision de l'analyse de cache :

– La première situation se produit lorsque le nombre de défauts de cache dans le L1 calculé statiquement est très proche de la valeur mesurée (*jfdctint*). Pour ce programme de test, l'analyse statique du cache L1 est très précise en

Benchmark	**Métriques**	**Analyse statique** small-32-32	**Mesure** small-32-32	**Analyse statique** small-32-64	**Mesure** small-32-64
jfdctint	nb d'accès L1	8039	8039	8039	8039
	nb de défauts L1	725	723	725	723
	nb de défauts L2	101	96	54	49
bs	nb d'accès L1	196	196	196	196
	nb de défauts L1	16	11	16	11
	nb de défauts L2	16	11	15	6
minver	nb d'accès L1	4146	4146	4146	4146
	nb de défauts L1	150	140	150	140
	nb de défauts L2	150	140	108	71
ns	nb d'accès L1	26428	26411	26428	26411
	nb de défauts L1	23	13	23	13
	nb de défauts L2	23	13	20	7
matmult	nb d'accès L1	525894	525894	525894	525894
	nb de défauts L1	51	41	51	41
	nb de défauts L2	51	38	49	19

Métrique a. Le nombre d'accès et le nombre de défauts de cache.

Benchmark	**Métriques**	**Analyse statique** small-32-32	**Mesure** small-32-32	**Analyse statique** small-32-64	**Mesure** small-32-64
jfdctint	L1+L2, cycles	25389	24869	20689	20169
	L1, cycles	87789		87789	
bs	L1+L2, cycles	1956	1406	1856	906
	L1, cycles	1956		1956	
minver	L1+L2, cycles	20646	19546	16446	12646
	L1, cycles	20646		20646	
ns	L1+L2, cycles	28958	27841	28658	27241
	L1, cycles	28958		28958	
matmult	L1+L2, cycles	531504	530104	531304	528204
	L1, cycles	531504		531504	

Métrique b. La contribution au pire temps d'exécution des accès mémoire.

TAB. 2.5: Précision de l'analyse statique de cache multi-niveaux d'une hiérarchie disposant de la politique de gestion non-inclusive avec la politique de remplacement LRU (L1 : 1KB assoc. 4 ; L2 : 2KB assoc. 8).

raison de la structure de l'application (présence de gros blocs de base et peu de structures de contrôle). Par conséquence, les accès considérés pendant l'analyse statique du cache L2 sont très proches des accès se produisant effectivement lors de l'exécution. De ce fait, le nombre de défauts de cache du L2 déterminé par l'analyse statique est également proche du nombre de défauts de cache du L2 se produisant lors de l'exécution. La surestimation de la valeur estimée de la contribution des accès mémoire au pire temps d'exécution est par conséquent petite (2%[1]). Dans ce cas, la différence entre la valeur estimée et la valeur mesurée provient essentiellement du pessimisme introduit par la classification *Uncertain* de tous les accès dont la présence dans le L1 ne peut pas être garantie statiquement.

– La seconde situation se produit lorsque l'analyse statique au niveau du cache L1 est légèrement moins précise (taille des blocs de base plus petite, plus de structure de contrôle). Ce comportement se répercute au niveau de l'analyse du cache L2 et est amplifié par l'introduction des accès classifiés *Uncertain*. Dans ce cas, l'analyse multi-niveaux reste tout de même relativement précise : 8% en moyenne en utilisant *minver*, *matmult* et *ns*. La moins bonne précision est observée sur *bs* (71%). Ce pessimisme est lié à la classification *First-Miss* des accès effectués au sein d'une boucle, combiné avec le faible nombre d'itération de celle-ci (4). Néanmoins, un grand nombre d'accès détectés comme produisant un défaut de cache dans le L1 lors de son analyse sont détectés comme produisant un succès au niveau du L2. L'estimation du pire temps d'exécution est de ce fait plus petite et donc plus intéressante que lors que juste le L1 est considéré.

Lorsque les tailles des lignes de cache entre les deux niveaux sont différentes, le nombre de défauts de cache lors de l'exécution et lors de l'analyse statique est toujours inférieur aux valeurs obtenues avec des lignes de cache de mêmes tailles. Cette observation montre que la localité spatiale est capturée par le L2 lors de l'exécution et que l'analyse statique est également apte à détecter cette localité.

Pour les programmes de plus grande taille (*healt_monitoring* et *telecommand*) uniquement les résultats de l'analyse statique sont présentés dans le tableau 2.6. Comme la taille du code de ces tâches est plus grande que celle des programmes de test, nous utilisons les configurations *medium-32-32* et *medium-32-64*.

Les résultats montrent que dans les deux configurations de cache utilisées, l'analyse détecte de façon significative de la réutilisation au niveau du cache L2 et l'estimation du pire temps d'exécution ce trouve donc améliorée par rapport à une analyse considérant uniquement le cache L1.

[1]La surestimation est une moyenne des valeurs obtenues avec les deux configurations de cache considérées. La surestimation pour une configuration donnée est définie de la façon suivante : $(\frac{\text{Contribution Analyse Statique } L1+L2}{\text{Contribution Mesurée}} - 1) * 100$

Benchmark	**Métriques**	**LRU**	
		32B	**64B**
health monitoring	nb d'accès L1	80863964	80863963
	nb de défauts L1	3857	3854
	nb de défauts L2	1298	1208
	L1+L2, cycles	81032334	81023303
	L1, cycles	81288234	81287903
telecommand	nb d'accès L1	40145	40145
	nb de défauts L1	4264	4264
	nb de défauts L2	149	127
	L1+L2, cycles	97685	95485
	L1, cycles	509185	509185

Métrique a. Le nombre d'accès et le nombre de défauts de cache.
et
Métrique b. La contribution au pire temps d'exécution des accès mémoire.

TAB. 2.6: Résultats de l'analyse statique de cache multi-niveaux d'une hiérarchie disposant de la politique de gestion non-inclusive avec la politique de remplacement LRU (L1 : 8KB assoc. 4 ; L2 : 16KB assoc. 8).

En résumé, la précision de notre analyse de cache multi-niveaux est globalement dépendante de la précision de l'analyse de cache initiale. Bien que l'analyse du cache L1 soit relativement précise quand ce cache est considéré seul, le pessimisme de cette analyse se trouve propagé aux niveaux suivants, ce qui a pour effet de réduire la précision de l'analyse multi-niveaux. Néanmoins pour chacun des codes analysés : (*i*) le pessimisme causé par la prise en compte des accès incertains pour des raisons de sûreté est raisonnable, (*ii*) les résultats obtenus lors de l'analyse de la hiérarchie sont toujours meilleurs comparativement à une analyse considérant uniquement le cache L1 et des défauts de cache systématique au niveau du cache L2.

Résultat pour une hiérarchie constituée de 3 niveaux de caches

Nous évaluons maintenant la précision de notre analyse pour une hiérarchie constituée de 3 niveaux de caches. Le programme de test utilisé pour cette expérimentation est une concaténation des programmes de test simples regroupés à l'intérieur d'une boucle effectuant deux itérations. Cette concaténation permet de regrouper différentes structures de code au sein du même programme de test (code de contrôle avec de petits blocs de base, code de calculs avec des blocs de base de taille plus conséquente). La hiérarchie de cache utilisée dans ce paragraphe est *small-32-32* étendue avec un troisième niveau disposant d'un degré d'associativité de 16 avec une taille de ligne de cache de 32 octets et une latence d'accès de 30 cycles. L'étude est effectuée sur deux tailles de cache pour le L3 : 4KB et 16KB pour que le programme loge dans le L3 dans la plus grande configuration mais pas dans la plus petite. Les résultats sont présentés dans le tableau 2.7.

La boucle externe de ce programme de test dispose d'une taille de code supé-

Taille du cache L3	Métriques	Analyse statique	Mesure
4 KB	nb d'accès L1	1129430	1129425
	nb de défauts L1	8669	1852
	nb de défauts L2	5236	608
	nb de défauts L3	1217	599
16 KB	nb d'accès L1	1129430	1129425
	nb de défauts L1	8669	1852
	nb de défauts L2	5236	608
	nb de défauts L3	348	298

Métrique a. Le nombre d'accès et le nombre de défauts de cache.

TAB. 2.7: Précision de l'analyse statique de cache multi-niveaux d'une hiérarchie disposant de la politique de gestion non-inclusive avec la politique de remplacement LRU (L1 : 1KB assoc. 4 ; L2 : 2KB assoc. 8 ; L3 : 4/16KB assoc. 16).

rieure à la capacité du cache L2 ce qui dégrade la précision de l'analyse *Persistence* au niveau du L1 et du L2 même en présence de boucles internes. Ce comportement a été identifié et solutionné dans [9] par une analyse de *Persistence* appliquée à chaque niveau de boucle.

Nous avons préalablement observé que lorsque l'analyse du cache L1 surestime le nombre de défaut de cache de ce niveau, l'impact était répercuté au niveau de la précision de l'analyse du cache L2. Ce comportement est également observable au niveau de l'analyse du L3. Cependant, nous pouvons remarquer que l'analyse du L3 permet tout de même de détecter de la réutilisation des informations contenue dans le L3. De ce fait, même avec une surestimation de 21,9% lorsque le L3 est de 4KB et de 17,7% lorsque le L3 est de 16KB, prendre en compte l'ensemble de la hiérarchie lors de l'analyse reste toujours meilleur comparativement à une analyse considérant uniquement le cache L1 et des défauts de cache systématiques dans les autres niveaux de la hiérarchie.

2.8.2.2 Résultats pour la politique de gestion *non-inclusive* disposant d'une politique de remplacement de cache non-LRU

L'impact de la politique de remplacement de cache sur la contribution de la hiérarchie mémoire à l'estimation du pire temps d'exécution est présenté dans le tableau 2.8 pour les configuration de cache *small-32-32* et *small-32-64*. Pour chaque programme de test, les résultats sont présentés pour chacune des politiques de remplacement de cache (LRU, PLRU, MRU, FIFO, Random). Comme expliqué précédemment, aucune mesure n'est effectuée en raison des effets dominos rendant l'exécution des tâches dans leur pire scénario difficile à réaliser.

Comme attendu d'après les résultats théoriques [86], la contribution des accès mémoire à l'estimation du pire temps d'exécution augmente lorsque la prévisibilité

Bench.	Métriques	LRU		PLRU		MRU		FIFO		RANDOM	
		32B	64B	32B	64B	32B	64B	32B	64B	32B	64B
matmult	nb d'accès L1	525894	525894	525894	525894	525894	525894	525894	525894	525894	525894
	nb de défauts L1	51	51	51	51	51	51	38971	38971	38971	38971
	nb de défauts L2	51	49	51	49	51	49	38971	38970	38971	38970
	L1+L2 ,cycles	531504	531304	531504	531304	531504	531304	4812704	4812604	4812704	4812604
	L1,cycles	531504	531504	531504	531504	531504	531504	4812704	4812704	4812704	4812704
jfdctint	nb d'accès L1	8039	8039	8039	8039	8039	8039	8039	8039	8039	8039
	nb de défauts L1	725	725	725	725	725	725	725	725	725	725
	nb de défauts L2	101	54	725	724	725	724	725	724	725	724
	L1+L2,cycles	25389	20689	87789	87689	87789	87689	87789	87689	87789	87689
	L1,cycles	87789	87789	87789	87789	87789	87789	87789	87789	87789	87789
bs	nb d'accès L1	196	196	196	196	196	196	196	196	196	196
	nb de défauts L1	16	16	16	16	16	16	16	16	16	16
	nb de défauts L2	16	15	16	15	16	15	16	15	16	15
	L1+L2,cycles	1956	1856	1956	1856	1956	1856	1956	1856	1956	1856
	L1,cycles	1956	1956	1956	1956	1956	1956	1956	1956	1956	1956
minver	nb d'accès L1	4146	4146	4146	4146	4146	4146	4146	4146	4146	4146
	nb de défauts L1	150	150	150	150	150	150	282	282	282	282
	nb de défauts L2	150	108	150	108	150	108	282	240	282	240
	L1+L2,cycles	20646	16446	20646	16446	20646	16446	35166	30966	35166	30966
	L1,cycles	20646	20646	20646	20646	20646	20646	35166	35166	35166	35166
ns	nb d'accès L1	26428	26428	26428	26428	26428	26428	26428	26428	26411	26411
	nb de défauts L1	23	23	23	23	23	23	2652	2652	2652	2652
	nb de défauts L2	23	20	23	20	23	20	2652	2649	2652	2649
	L1+L2,cycles	28958	28658	28958	28658	28958	28658	318148	317848	318148	317848
	L1,cycles	28958	28958	28958	28958	28958	28958	318148	318148	318148	318148
crc	nb d'accès L1	141643	141643	141643	141643	141643	141643	141655	141655	141671	141671
	nb de défauts L1	101	101	101	101	101	101	18599	18599	18599	18599
	nb de défauts L2	101	93	101	93	101	93	18599	18591	18599	18591
	L1+L2,cycles	152753	151953	152753	151953	152753	151953	2187545	2186745	2187545	2186745
	L1,cycles	152753	152753	152753	152753	152753	152753	2187545	2187545	2187545	2187545
qurt	nb d'accès L1	6688	6688	6688	6688	6691	6691	6691	6691	6691	6691
	nb de défauts L1	163	163	163	163	655	655	931	931	931	931
	nb de défauts L2	163	147	163	147	655	639	931	915	931	915
	L1+L2,cycles	24618	23018	24618	23018	78741	77141	109101	107501	109101	107501
	L1,cycles	24618	24618	24618	24618	78741	78741	109101	109101	109101	109101
fft	nb d'accès L1	80305	80305	80310	80310	80310	80310	80310	80310	80310	80310
	nb de défauts L1	6687	6687	9299	9299	11096	11096	12342	12342	12342	12342
	nb de défauts L2	326	315	7229	6671	11096	11085	12342	12333	12342	12333
	L1+L2,cycles	179775	178675	896200	840400	1300870	1299770	1437930	1437030	1437930	1437030
	L1,cycles	815875	815875	1103200	1103200	1300870	1300870	1437930	1437930	1437930	1437930
adpcm	nb d'accès L1	187395	187395	187395	187395	187395	187395	184396	184396	184396	184396
	nb de défauts L1	3907	3907	9923	9923	16085	16085	28312	28312	28312	28312
	nb de défauts L2	3907	3126	3907	3899	12931	15714	28312	28304	28312	28304
	L1+L2,cycles	617165	539065	677325	676525	1641345	1919645	3298716	3297916	3298716	3297916
	L1,cycles	617165	617165	1278925	1278925	1956745	1956745	3298716	3298716	3298716	3298716
statem.	nb d'accès L1	10931	11011	10931	11011	10931	11011	10931	11011	10931	10931
	nb de défauts L1	1815	1805	1815	1815	1835	1825	1875	1865	1875	1875
	nb de défauts L2	1802	1160	1815	1802	1835	1492	1875	1533	1875	1863
	L1+L2,cycles	209281	145061	210581	209361	212781	178461	217181	182961	217181	215981
	L1,cycles	210581	209561	210581	209561	212781	211761	217181	216161	217181	217181

Métrique a. Le nombre d'accès et le nombre de défauts de cache.
et
Métrique b. La contribution au pire temps d'exécution des accès mémoire.

TAB. 2.8: Impact sur l'estimation du pire temps d'exécution de la politique de remplacement de cache d'une hiérarchie disposant de la politique de gestion non-inclusive (L1 : 1KB assoc 4 ; L2 : 2KB assoc 8).

de la politique de remplacement de cache diminue (i.e. $LRU \preceq PLRU \preceq MRU \preceq FIFO \preceq RANDOM$). De plus, de façon similaire à la politique LRU, nous observons que le pessimisme de l'analyse du cache L1 est propagé au niveau suivant. En effet, plus la précision de l'analyse de cache L1 est bonne pour une politique de remplacement de cache donnée, plus la précision de l'analyse du cache L2 est bonne.

Néanmoins, l'impact de la politique de remplacement de cache sur l'estimation du pire temps d'exécution est différente suivant la structure du code analysé, en particulier la taille des boucles comparée à la taille du cache. Pour *adpcm*, la contribution des accès mémoire à l'estimation du pire temps d'exécution augmente de façon régulière lorsque la politique de remplacement de cache est moins prévisible. *statemate* a le même comportement sauf dans le cas de la politique de remplacement de cache PLRU avec la configuration de cache *small-32-64*. Dans cette configuration, la borne *evict* entraîne de nombreux accès classifiés comme incertain lors de l'analyse du cache L1 dégradant alors les performances de l'analyse du cache L2. Pour d'autres programmes de test, nous observons un seuil de dégradation significatif entre LRU et PLRU (*jfdctint* et *fft*), entre PLRU et MRU (*qurt*) ou encore entre MRU et FIFO (*matmult*, *minver*, *ns* et *crc*). Pour *bs*, la politique de remplacement de cache n'influence pas les résultats de l'analyse car la taille des boucles loge dans le L1 même dans le cas de où la taille du cache abstrait est réduite par la borne *mls*. Pour les tâches de l'application réelle, les résultats présentés dans le tableau 2.9 montrent le même type de comportement qu'avec les programmes de test.

Bench.	Métriques	LRU		PLRU		MRU		FIFO/RANDOM	
		32B	64B	32B	64B	32B	64B	32B	64B
health monitoring	nb d'accès L1	80863964	80863963	80872547	80863997	80864208	80864186	78523506	78523505
	nb de défauts L1	3857	3854	9027	8459	2893566	2893563	9198209	9198206
	nb de défauts L2	1298	1208	9027	8374	2759438	2886339	9198209	9198128
	L1+L2,cycles	81032334	81023303	81865517	81785987	385743668	398433716	1090326496	1090318365
	L1,cycles	81288234	81287903	81865517	81794487	399156468	399156116	1090326496	1090326165
telecom.	nb d'accès L1	40145	40145	40188	40188	40188	40188	40188	40188
	nb de défauts L1	4264	4264	5296	5296	5300	5300	5303	5303
	nb de défauts L2	149	127	5295	4763	5300	5278	5303	5281
	L1+L2,cycles	97685	95485	622648	569448	623188	620988	623518	621318
	L1,cycles	509185	509185	622748	622748	623188	623188	623518	623518

Métrique a. Le nombre d'accès et le nombre de défauts de cache.
et
Métrique b. La contribution au pire temps d'exécution des accès mémoire.

TAB. 2.9: Impact sur l'estimation du pire temps d'exécution de la politique de remplacement de cache d'une hiérarchie disposant de la politique de gestion non-inclusive (L1 : 8KB assoc 4 ; L2 : 16KB assoc 8).

La connaissance de ce comportement pour une tâche donnée peut être très utile pendant la phase de conception du système afin de trouver le meilleur compromis entre la prévisibilité et le coût du système, car la politique moins prévisible est généralement également moins coûteuse.

Impact de la classification d'accès au cache *Uncertain*

L'intérêt principal de la classification d'accès au cache *Uncertain*, comme expliqué dans la section 2.3.2, est de modéliser le filtrage des accès effectué par chaque niveau de cache composant la hiérarchie dès lors que l'analyse statique ne peut pas garantir un succès ou un défaut de cache dans un niveau ℓ. La classification *Uncertain* garantit alors la sûreté de l'analyse du niveau $\ell+1$. Cependant, cette classification peut également être utilisée pour l'ensemble des accès à un niveau de cache donné ce qui est requis pour l'analyse des hiérarchies de caches disposant d'une politique de gestion *inclusive* mais ce qui aussi permettre de paralléliser l'exécution des analyses de chacun des niveaux composant la hiérarchie.

Évidemment, considérer tous les accès à un niveau de cache comme incertains dégrade la précision de l'analyse en raison de la perte d'information sur les références mémoire qui accède un niveau donné. Nous évaluons cette perte de précision en utilisant les configurations précédentes et en considérant tous les accès aux L2 comme incertains. La métrique utilisée est l'amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution avec d'une part une analyse ayant tous les accès au cache L2 classifiés *Uncertain* et d'autre part l'analyse originale des hiérarchies non-inclusive ($\frac{Analyse_{Uncertain}}{Analyse_{NI}} - 1$). Les résultats sont fournis dans le tableau 2.10 pour chacune des politiques de remplacement de cache.

Bench.	LRU		PLRU		MRU		FIFO		RANDOM	
	32B	64B	32B	64B	32B	64B	32B	64B	32B	64B
matmult	0.00 %	0.04 %	0.00 %	0.04 %	0.00 %	0.04 %	0.00 %	0.00 %	0.00 %	0.00 %
jfdctint	0.00 %	22.72 %	0.00 %	0.11 %	0.00 %	0.11 %	0.00 %	0.11 %	0.00 %	0.11 %
bs	0.00 %	5.39 %	0.00 %	5.39 %	0.00 %	5.39 %	0.00 %	5.39 %	0.00 %	5.39 %
minver	0.00 %	25.54 %	0.00 %	25.54 %	0.00 %	25.54 %	0.00 %	13.56 %	0.00 %	13.56 %
ns	0.00 %	1.05 %	0.00 %	1.05 %	0.00 %	1.05 %	0.00 %	0.09 %	0.00 %	0.09 %
crc	0.00 %	0.53 %	0.00 %	0.53 %	0.00 %	0.53 %	0.00 %	0.04 %	0.00 %	0.04 %
qurt	0.00 %	6.95 %	0.00 %	6.95 %	0.00 %	2.07 %	0.00 %	1.49 %	0.00 %	1.49 %
fft	0.00 %	0.62 %	0.00 %	0.13 %	0.00 %	0.08 %	0.00 %	0.06 %	0.00 %	0.06 %
adpcm	0.00 %	14.49 %	0.00 %	0.12 %	19.22 %	1.93 %	0.00 %	0.02 %	0.00 %	0.02 %
statemate	0.62 %	45.17 %	0.00 %	0.62 %	0.00 %	19.23 %	0.00 %	18.70 %	0.00 %	0.56 %

Métriques c. Amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution. ($\frac{Analysis_{Uncertain}}{Analysis_{NI}} - 1$).

Bench.	LRU		PLRU		MRU		FIFO		RANDOM	
	32B	64B	32B	64B	32B	64B	32B	64B	32B	64B
health monitoring	0.00 %	0.01 %	0.00 %	0.01 %	3.48 %	0.09 %	0.00 %	0.00 %	0.00 %	0.00 %
telecommand	0.00 %	2.30 %	0.02 %	9.36 %	0.00 %	0.35 %	0.00 %	0.35 %	0.00 %	0.35 %

Métriques c. Amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution. ($\frac{Analysis_{Uncertain}}{Analysis_{NI}} - 1$).

TAB. 2.10: Impact sur l'estimation du pire temps d'exécution de la classification d'accès au cache *Uncertain* avec les différentes politiques de remplacement de cache et une hiérarchie disposant de la politique de gestion non-inclusive (Table du haut : L1 1KB assoc 4 ; L2 2KB assoc 8 / Table du bas : L1 8KB assoc 4 ; L2 16KB assoc 8).

Deux comportements distincts sont observés en fonction de la taille des lignes de cache du L2. Lorsque la taille des lignes de cache est égale pour les deux ni-

veaux, la surestimation résultante est dans la plupart des cas de 0%. En d'autres termes, la précision de l'analyse n'est pas fortement impactée par les accès classifiés *Uncertain*. Dans cette configuration, le cache L2 capture uniquement la localité temporelle lors de l'exécution. Cette localité temporelle est également bien capturée par l'analyse *Persistence* qui ne souffre que très légèrement de l'introduction des accès classifiés *Uncertain*.

Le second comportement est observé lorsque la taille des lignes de cache du L2 est supérieure à celles du L1. Dans cette configuration, le L2 capture la localité spatiale et temporelle lors de l'exécution mais comme nous l'avons vu précédemment, l'analyse est quand elle incapable de détecter la localité spatiale. Dans ce cas, la précision de l'analyse souffre de l'introduction des accès classifiés *Uncertain* de façon significative (4.89% en moyenne pour les différentes politiques de remplacement de cache et l'ensemble des programmes de test et 45.17% au maximum pour *statemate* avec du LRU). Ce second comportement montre l'intérêt d'avoir une classification d'accès au cache précise pour l'analyse des hiérarchies de cache non-inclusive dès lors que les tailles des lignes de cache des différents niveaux sont différentes.

2.8.2.3 Résultats pour la politique de gestion inclusive

Pour garantir la propriété d'inclusion, l'invalidation d'une ligne de cache peut se produire dans un niveau de cache lorsque celle si est évincée d'un niveau de cache inférieur de la hiérarchie. Ce comportement est difficile à prédire précisément et peut amener un pessimisme supplémentaire lors de l'analyse, comparativement à l'analyse de hiérarchies de caches non-inclusives. Nous avons observé, en exécutant les programmes de test dans leur pire scénario d'exécution en considérant deux niveaux de cache disposant de la politique de remplacement de cache LRU, que le nombre de défauts de cache est identique pour une hiérarchie de cache inclusive ou non-inclusive. De ce fait, nous proposons d'évaluer l'amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution entre les résultats de l'analyse statique pour la politique de gestion inclusive (I) et non-inclusive (NI) ($\frac{Analysis_I}{Analysis_{NI}} - 1$). Les résultats sont présentés dans le tableau 2.11.

Intéressons nous dans un premier temps à la politique de remplacement de cache LRU prouvée [86] comme étant la plus apte à être prédite précisément. Les résultats montrent que lorsque les tailles des lignes de cache sont identiques pour les deux niveaux, le pessimisme résultant de l'ajout du mécanisme d'inclusion est généralement relativement faible excepté pour *adpcm* et *statemate*. Ces derniers disposent de boucles de taille supérieure à la capacité du cache L2 amenant alors l'analyse à considérer de nombreuses invalidations. Lorsque les lignes de cache sont de taille différente entre les niveaux, nous observons, grâce au post traitement

Bench.	LRU		PLRU		MRU		FIFO		RANDOM	
	32B	64B	32B	64B	32B	64B	32B	64B	32B	64B
matmult	0.00 %	-0.23 %	0.00 %	-0.23 %	0.00 %	3.65 %	41.50 %	25.13 %	41.50 %	25.13 %
jfdctint	0.00 %	0.48 %	2.13 %	-38.12 %	2.13 %	-38.12 %	2.13 %	-38.12 %	2.13 %	-38.12 %
bs	0.00 %	-16.16 %	0.00 %	-16.16 %	0.00 %	-16.16 %	0.00 %	-16.16 %	0.00 %	-16.16 %
minver	1.94 %	-5.35 %	1.94 %	-5.35 %	1.94 %	-5.35 %	6.88 %	-12.11 %	6.88 %	-12.11 %
ns	0.00 %	-0.70 %	0.00 %	-0.70 %	0.00 %	-0.70 %	35.44 %	19.06 %	35.44 %	19.06 %
crc	0.00 %	-1.78 %	59.17 %	61.32 %	123.57 %	108.95 %	87.99 %	42.40 %	87.99 %	42.40 %
qurt	0.00 %	-14.77 %	41.03 %	38.23 %	47.00 %	44.40 %	36.98 %	14.17 %	36.98 %	14.17 %
fft	0.00 %	-5.88 %	21.20 %	44.49 %	23.51 %	-5.25 %	15.48 %	-12.95 %	15.48 %	-12.95 %
adpcm	29.89 %	15.88 %	27.24 %	243.06 %	150.95 %	56.14 %	57.40 %	23.85 %	57.40 %	23.85 %
statem.	41.62 %	63.43 %	40.80 %	13.38 %	39.34 %	32.97 %	36.52 %	29.69 %	36.52 %	9.87 %

Métriques c. Amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution. ($\frac{Analysis_I}{Analysis_{NI}} - 1$).

Bench.	LRU		PLRU		MRU		FIFO		RANDOM	
	32B	64B	32B	64B	32B	64B	32B	64B	32B	64B
health mon.	0.42 %	1.41 %	329.49 %	327.49 %	142.18 %	159.21 %	129.90 %	89.70 %	129.90 %	89.70%
telecommand	0.00 %	-2.72 %	64.36 %	41.78 %	64.27 %	30.10 %	64.18 %	30.03 %	64.18 %	30.03 %

Métriques c. Amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution. ($\frac{Analysis_I}{Analysis_{NI}} - 1$).

TAB. 2.11: Comparaison entre la politique de gestion non-inclusive et inclusive (Table du haut : L1 1KB assoc 4 ; L2 2KB assoc 8 / Table du bas : L1 8KB assoc 4 ; L2 16KB assoc 8).

basé sur la propriété d'inclusion réalisé au niveau du cache L2, une diminution de l'estimation du pire temps d'exécution comparativement à une hiérarchie de cache non-inclusive. Ce comportement n'est toute fois pas systématique car dans certains cas, le pessimisme résultant de la classification *Uncertain* cumulé à la prise en compte des invalidations n'est pas compensé par ce post traitement et nous observons dans ce cas une dégradation de l'estimation du pire temps d'exécution.

De façon similaire à l'analyse des hiérarchies de caches non-inclusives avec les politiques de remplacement de cache non-LRU, nous observons que globalement le pessimisme de l'analyse augmente lorsque la prévisibilité de la politique de remplacement de cache diminue, lorsque les tailles des lignes de cache des deux niveaux sont identiques. Par contre, lorsque les tailles sont différentes, ce comportement est moins uniforme en raison du post traitement qui permet de réduire le pessimisme, voire d'améliorer l'estimation suivant les cas en fonction du nombre d'invalidations considérées lors de l'analyse comparativement au gain du post traitement.

En conclusion, il s'avère préférable d'utiliser une hiérarchie de caches non-inclusive lorsque les tailles des lignes de cache entre les niveaux sont identiques. Lorsque les tailles de lignes sont différentes, le choix est plus dépendant de la structure du code, notamment la taille des boucles comparativement à la taille du cache L2, entraînant dans certains cas l'analyse à considérer un nombre important d'invalidations pouvant ne pas être compensées par le post-traitement exploitant la propriété d'inclusion.

2.8.2.4 Résultats pour la politique de gestion exclusive

Afin de déterminer la précision de l'analyse des hiérarchies de caches exclusives, nous comparons les résultats de l'analyse statique avec ceux obtenus lors de l'exécution des programmes de test dans leur pire scénario d'exécution. Les résultats sont présentés dans le tableau 2.12 pour les programmes de test à chemin unique avec la configuration *small-32-32* disposant de la politique de remplacement LRU. Pour chaque programme de test, nous donnons trois valeurs : la contribution des accès mémoire au pire temps d'exécution déterminée par l'analyse statique (ligne du haut) ainsi que celle obtenue par mesure sur le pire chemin (ligne du milieu) et la surestimation de l'analyse ($\frac{analyse}{mesure} - 1$, ligne du bas). Ces trois valeurs sont données respectivement pour une hiérarchie de caches exclusive (colonne de gauche) et pour une hiérarchie de caches non-inclusive (colonne de droite) celle-ci étant toujours plus précise dans cette configuration par rapport à une hiérarchie de cache inclusive.

jfdctint	Exclusive	Non-Inclusive
Analyse statique	24989	25389
Mesure	24769	24869
Surestimation (%)	0.89	2.09
bs	Exclusive	Non-Inclusive
Analyse statique	1956	1956
Mesure	1406	1406
Surestimation (%)	39.12	39.12
minver	Exclusive	Non-Inclusive
Analyse statique	20646	20646
Mesure	19546	19546
Surestimation (%)	5.63	5.63
ns	Exclusive	Non-Inclusive
Analyse statique	28958	28958
Mesure	27841	27841
Surestimation (%)	4.01	4.01
matmult	Exclusive	Non-Inclusive
Analyse statique	531394	531504
Mesure	530104	530104
Surestimation (%)	0.24	0.26

TAB. 2.12: Comparaison de la précision entre la politique de gestion non-inclusive et exclusive (L1 : 1KB assoc 4 ; L2 : 2KB assoc 8).

La première observation intéressante concernant les résultats de l'analyse de hiérarchies de caches exclusives est que les valeurs obtenues sont généralement proches de celles obtenues lors de l'exécution. Un pessimisme raisonnable entre

0.24% (*matmult*) et 5.63% (*minver*) est observé, sauf pour *bs* en raison de son faible nombre d'accès, qui amplifie l'impact des défauts de cache déterminés statiquement mais ne se produisant pas lors de l'exécution.

Si nous comparons maintenant la surestimation de l'analyse comparativement à l'exécution pour les hiérarchies de cache exclusives et non-inclusives, nous observons que la surestimation obtenue pour les hiérarchies exclusives est toujours inférieure ou égale à celle des hiérarchies non-inclusive. Pour *jfdctint*, la différence est significative et s'explique par l'absence d'accès considérés comme incertains dans l'analyse améliorant ainsi sa précision. Finalement pour ce programme de test, nous observons également que le pire temps mesuré est meilleur dans le cas d'une hiérarchie exclusive car ce type de hiérarchie permet de mieux exploiter sa localité.

Benchmark	**LRU**	**PLRU**	**MRU**	**FIFO**	**RANDOM**
matmult	-0.02 %	-0.02 %	-0.02 %	-88.96 %	-88.96 %
jfdctint	-1.58 %	-71.54 %	0.00 %	0.00 %	0.00 %
bs	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
minver	0.00 %	0.00 %	0.00 %	-41.29 %	-41.29 %
ns	0.00 %	0.00 %	0.00 %	-90.90 %	-90.90 %
crc	0.00 %	0.00 %	0.00 %	-93.02 %	-93.02 %
qurt	0.00 %	0.00 %	-68.74 %	-27.83 %	-27.83 %
fft	-0.11 %	-79.96 %	-37.28 %	-9.53 %	-9.53 %
adpcm	-10.94 %	-8.88 %	-62.40 %	-40.68 %	-40.68 %
statemate	0.00 %	-0.09 %	-1.03 %	-2.03 %	-2.03 %

Métriques c. Amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution. ($\frac{Analysis_E}{Analysis_{NI}} - 1$).

Benchmark	**LRU**	**PLRU**	**MRU**	**FIFO**	**RANDOM**
health monitoring	0.00 %	-1.02 %	-78.93 %	-63.39 %	-63.39 %
telecommand	0.00 %	-76.07 %	-18.25 %	-0.05 %	-0.05 %

Métriques c. Amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution. ($\frac{Analysis_E}{Analysis_{NI}} - 1$).

Tab. 2.13: Comparaison entre la politique de gestion non-inclusive et exclusive (Table du haut : L1 1KB assoc 4 ; L2 2KB assoc 8 / Table du bas : L1 8KB assoc 4 ; L2 16KB assoc 8).

Comparons maintenant les résultats des analyses des hiérarchies exclusives et non-inclusives pour chacune des politiques de remplacement de cache. Afin d'évaluer les bénéfices de la hiérarchie exclusive (E), nous comparons les résultats obtenue avec ceux d'une hiérarchie non-inclusive (NI) en utilisant la métrique d'amélioration/dégradation relative de la contribution des accès mémoire au pire temps d'exécution ($\frac{Analyse_E}{Analyse_{NI}} - 1$). Les résultats sont présentés dans le tableau 2.13.

La première observation générale est que l'estimation de la contribution des accès mémoire au pire temps d'exécution est toujours inférieure ou égale à l'estimation obtenue pour une hiérarchie non-inclusive. Ce comportement est lié à la

nature des hiérarchies exclusives, qui permet d'augmenter virtuellement la capacité mémoire et l'analyse est à même de le prendre en compte. Pour une majorité de programmes de test, il y a même un écart important entre les deux estimations pour au moins une politique de remplacement de cache. Par exemple, pour *matmult* avec la politique FIFO, nous observons une amélioration de 88.96%. Cette différence s'explique par la borne *mls* (1 pour FIFO) qui réduit virtuellement la capacité de la hiérarchie lors de l'analyse. Cependant, la non duplication d'information dans le cas de l'exclusif permet de limiter cette réduction et ainsi de déterminer une meilleur estimation. Pour les autres programmes de test, le même comportement est observé et à chaque fois, le phénomène se produit avec la politique de remplacement de cache détectée comme produisant une différence significative dans l'estimation du pire temps d'exécution en comparaison à une politique plus prévisible (section 2.8.2.2).

En conclusion, l'utilisation de hiérarchies de caches exclusives est préférable, en comparaison des hiérarchies non-inclusives, pour produire une estimation plus fine du pire temps d'exécution et tout spécialement lorsque des politiques de cache non-LRU sont utilisées.

2.9 Conclusion

Dans ce chapitre, nous venons de présenter les méthodes d'analyse statique de hiérarchies de caches d'instructions publiées dans [40, 42]. Nous avons commencé par mettre en avant un problème de sûreté dans l'unique approche existante [68] provenant de la propagation systématique des accès incertains. Nous avons ensuite proposé des méthodes d'analyse statique pour les hiérarchies de caches disposant des politiques de gestion non-inclusive, inclusive et exclusive. Ces approches modélisent, grâce à l'introduction d'une nouvelle classification appelée classification d'accès au cache, et prennent en compte de façon sûre le comportement des accès incertains dans les différents niveaux de la hiérarchie lors de l'analyse. Nous avons vu que ces méthodes sont compatibles avec les politiques de remplacement de cache LRU, Pseudo-LRU, MRU, FIFO et Random en se basant sur une adaptation des résultats théoriques proposés dans [86]. Les résultats expérimentaux montrent que globalement l'analyse des hiérarchies de caches permet de raffiner l'estimation du pire temps d'exécution comparativement aux méthodes existantes ne considérant qu'un seul niveau de cache et des défauts de cache systématiques dans les niveaux inférieurs de la hiérarchie.

En réponse à la première question de recherche de ce document, nous avons montré, dans ce chapitre, la faisabilité d'analyser des hiérarchies de caches de façon sûre. Concernant la précision de l'estimation, elle est différente suivant le type de hiérarchie considéré mais toujours plus précise que l'estimation produite par des analyses ne considérant qu'un seul niveau de cache.

Nous avons étendu l'analyse des hiérarchies de caches disposant d'une politique de gestion non-inclusive aux données dans [49]. La modification principale, pour pouvoir analyser les accès aux données, consiste à modifier la fonction *Update* afin d'intégrer le fait qu'une instruction accédant à la mémoire peut référencer différentes adresses mémoires. Cette fonction ainsi modifiée est ensuite directement applicable pour analyser les différentes politiques de gestion de hiérarchies étudiées dans ce chapitre.

Notre méthode a également été étendue dans [22] pour prendre en compte les caches unifiés. Finalement, cette méthode que nous utiliserons comme base, dans le chapitre suivant, pour analyser les hiérarchies de caches des architectures multi-cœurs, a également été utilisée pour les mêmes raisons dans [52].

Généralement, les hiérarchies de caches disposant de plus de deux niveaux de cache mettent en œuvre une combinaison des différentes politiques de gestion de la hiérarchie. Par exemple, l'AMD Opteron 2384, l'Intel Xeon X5570 et l'IBM Power5 mettent en œuvre trois combinaisons différentes. L'AMD Opteron 2384 met en œuvre une politique d'exclusion entre le cache de premier et de second niveau tandis que le dernier niveau est non-inclusif. L'Intel Xeon X5570 met en œuvre une politique non-inclusive entre les deux premiers niveaux et une politique inclusive pour le dernier niveau. Finalement, l'IBM Power5 met en œuvre une politique inclusive entre les deux premier niveaux tandis que le dernier niveau est exclusif par rapport aux deux premiers. Une extension possible de la méthode d'analyse proposée serait d'étendre l'analyse à des hiérarchies de caches ne disposant pas d'une politique de gestion unique entre tous les niveaux de la hiérarchie.

Chapitre 3

Analyse du contenu pire cas de caches partagés pour architecture multi-cœurs

3.1 Introduction

Pendant de nombreuses années, le gain en performance obtenu entre deux générations de processeurs reposait principalement sur l'augmentation de la fréquence d'horloge du processeur. Ainsi, nous sommes passés d'une fréquence de quelques mégahertz, 4,77 MHz pour le processeur Intel 8088 commercialisé avec le premier ordinateur personnel d'IBM au début des années 80, à quelques gigahertz, 3,8 GHz pour le processeur Intel Pentium IV Prescott proposé au printemps 2004. Cette approche a néanmoins montré certaines limites surtout en terme de consommation et de dissipation thermique. Ce point devenant crucial, les fabricants de processeurs ont pris une nouvelle direction pour améliorer les performances des processeurs actuels, celle de mettre plusieurs cœurs de calcul sur une même puce permettant alors l'exécution simultanée de plusieurs applications. Ce type d'architecture, communément appelé multi-cœurs, se retrouve de nos jours dans les ordinateurs de bureau, les serveurs et est envisagé à moyen terme dans les systèmes embarqués comme ceux utilisés par l'industrie automobile [6].

Différents types de processeurs multi-cœurs ont vu le jour. Ceux disposant de cœurs homogènes, c'est-à-dire avec tous leurs cœurs identiques comme les processeurs Intel dual-core, quad-core, AMD Phenom, Sun Niagara, ARM11MPcore... et ceux disposant de cœurs hétérogènes comme le processeur CELL d'IBM. Vis-à-vis de la hiérarchie caches, plusieurs possibilités ont également été explorées [45]. Généralement, chaque cœur dispose d'un premier niveau de cache privé tandis que les niveaux suivants peuvent être partagés, semi-partagés ou privés. Les hiérarchies disposant d'un niveau de cache partagé entre les différents cœurs de calcul,

comme le Power5 d'IBM ou le Core Duo d'Intel, permettent le partage de données entre applications ce qui améliore généralement le temps d'exécution moyen des applications utilisant plusieurs processus légers. Elles offrent également la possibilité pour une application d'utiliser l'intégralité du cache partagé dans le cas où l'application est exécutée seule sur le processeur. Cependant, ce type de hiérarchie introduit également un nouveau type de défaut de cache : les défauts de conflits inter-tâches. En effet, le cache étant partagé, le chargement d'une ligne de cache par une application exécutée sur un cœur peut évincer une ligne de cache d'une autre application exécutée simultanément sur un autre cœur. En opposition, les hiérarchies disposant de caches uniquement privés, comme l'Athlon X2 d'AMD ou le Power6 d'IBM, ne souffrent pas des défauts de conflits inter-tâches dans les niveaux de caches inférieurs ce qui améliore généralement le temps moyen d'exécution d'applications indépendantes exécutées simultanément. Cependant, la capacité d'utilisation du cache est dans ce cas limitée à la taille du cache privé associé à un cœur de calcul. Finalement, les hiérarchies semi-partagées peuvent être vues comme un compromis entre les deux types de hiérarchies précédentes car elles partagent des niveaux de caches mais uniquement entre un sous-ensemble des cœurs composant le processeur, comme l'Intel Dunnington ou Harpertown.

L'utilisation des processeurs multi-cœurs dans le cadre des systèmes temps-réel amène de nouvelles problématiques liées aux ressources partagées (caches, bus mémoire) pour assurer la validation temporelle des systèmes. En effet, la présence de niveaux de caches partagés entre différents cœurs de calcul introduit une nouvelle source d'indéterminisme lors de l'estimation du pire temps d'exécution provoquée par les défauts de conflits inter-tâches. Pour prendre en compte, les hiérarchies disposant de caches partagés ou semi-partagés, l'analyse doit tenir compte de l'effet de ces défauts. Nous verrons que l'approche existante [110] visant à estimer les différents entrelacements possibles se limite à l'exécution de deux tâches sur deux cœurs de calcul et que dans certains cas elle peut amener à des problèmes de sûreté.

3.1.1 Contributions

Dans ce chapitre, nous nous intéressons à l'estimation du pire temps d'exécution d'une tâche exécutée sur un processeur multi-cœurs disposant de niveaux de caches partagés. L'approche proposée consiste à prendre en compte les défauts de conflits inter-tâches lors de cette estimation, tout en faisant abstraction des entrelacements possibles avec les tâches interférentes afin de minimiser la complexité et ainsi permettre l'analyse d'architectures disposant d'un nombre de cœurs quelconque exécutant un nombre arbitraire de tâches simultanément. Néanmoins, cette approche utilisée seule souffre d'un pessimisme potentiellement important en fonction du nombre de cœurs et de tâches considérés. Nous pro-

posons d'utiliser conjointement cette méthode d'estimation avec un mécanisme de court-circuit des caches partagés, appelé *bypass*, visant à réduire les interférences provoquées par chacune des tâches. L'idée consiste à limiter le stockage des lignes de cache dans les niveaux partagés en chargeant uniquement les lignes statiquement connues comme étant réutilisées lorsque la tâche est analysée en isolation. Ceci permet de réduire considérablement le nombre d'interférences sans produire d'effet négatif sur l'estimation du pire temps d'exécution de la tâche en isolation. Cette solution appliquée lors de la phase de compilation permet de réduire de façon importante les défauts de conflits inter-tâches permettant alors une estimation plus précise du pire temps d'exécution.

3.1.2 Organisation du chapitre

La suite de ce chapitre est organisée de la manière suivante. Nous commencerons (section 3.2) par étudier la méthode proposée dans [110] visant à estimer, par une analyse statique, le temps d'exécution pire cas d'une tâche, exécutée sur un processeur bi-cœurs disposant d'un niveau de cache partagé, en conflits avec une tâche s'exécutant simultanément. Nous verrons les limites de cette méthode vis-à-vis de la propriété de sûreté de l'estimation du pire temps d'exécution. Nous présenterons, par la suite (section 3.4), une méthode sûre permettant l'estimation des conflits, dans les niveaux de caches partagés, ainsi que leurs prises en compte lors de l'estimation du pire temps d'exécution. Cependant, cette approche gros grain peut rapidement mener l'analyse à ne détecter aucune réutilisation dans les niveaux de caches partagés. Nous aborderons ensuite (section 3.5) la contribution principale de ce chapitre, à savoir une approche exploitant un mécanisme matériel, appelé *bypass*, permettant de réduire le nombre de conflits dans ces niveaux de caches en évitant le stockage de certaines lignes de cache. Une étude expérimentale sera présentée dans la section 3.6 puis, nous conclurons ce chapitre en proposant des directions possibles pour les travaux futurs.

3.2 Approche existante

La première approche d'analyse statique visant à estimer le pire temps d'exécution d'une tâche exécutée sur un processeur multi-cœurs disposant d'un niveau de cache partagé a été proposée dans [110]. Cette approche se focalise sur l'analyse des caches d'instructions à correspondance directe en considérant une architecture constituée de deux cœurs de calcul disposant chacun d'un premier niveau de cache privé et d'un second niveau partagé. L'analyse se concentre sur l'estimation du pire temps d'exécution d'une tâche en conflit au niveau du cache partagé avec une autre tâche exécutée simultanément. Dans un premier temps, le comportement pire cas (succès ou défaut au niveau du cache partagé) de chaque accès est

estimé en considérant la tâche analysée en isolation puis dans un second temps, les accès classifiés comme produisant un succès sont éventuellement modifiés en défaut si la tâche concurrente peut être en interférence. Plus précisément, une distinction est faite si l'accès se produit ou non dans une boucle et si l'un des accès interférents est situé ou non dans une boucle. Si l'accès est considéré en dehors d'une boucle, la moindre interférence implique la modification de la classification en *défaut* tandis que si l'accès se produit au sein d'une boucle deux cas sont alors distingués :

- l'accès interférent est également situé dans une boucle, dans ce cas la classification de l'accès est modifiée en *défaut* ;
- aucun accès interférent n'est situé dans une boucle, dans ce cas la classification est modifiée en *succès sauf un* signifiant que l'interférence évincera une seule fois la ligne de cache contenant la référence accédée.

Cette approche permet de modéliser l'impact des différents entrelacements possibles entre deux tâches lors de l'estimation du pire temps d'exécution sans pour autant le faire de façon exhaustive. Néanmoins, la classification introduite *succès sauf un* peut dans certains cas mettre en défaut la sûreté de l'analyse. En effet, lors de la prise en compte des interférences, cette méthode ne considère pas le fait que plusieurs zones de la tâche interférente peuvent être exécutées pendant l'exécution de la boucle de la tâche analysée et donc évincer plusieurs fois une même ligne de cache. L'approche que nous proposons corrige ce problème de sûreté en faisant abstraction des entrelacements lors de l'analyse. De plus, notre approche est applicable pour des architectures constituées de plus de deux cœurs de calcul et disposant de plusieurs niveaux de caches partagés. Finalement, la structure de chaque niveau de cache est dans notre cas considérée comme étant associatif par ensembles, ceci étant plus représentatif des architectures existantes.

3.3 Hypothèses et notations

Dans cette partie, nous considérons une architecture multi-cœurs constituée de N cœurs de calcul. Chacun des cœurs dispose d'un cache d'instructions de premier niveau privé suivi par différents niveaux de caches dont au moins un est partagé entre plusieurs cœurs. Tous les caches sont associatifs par ensembles et nous considérons, lors de la description des méthodes que la politique de remplacement de cache LRU est mise en œuvre par chaque niveau de cache. Néanmoins, les autres politiques de remplacement de cache sont directement exploitables en utilisant les bornes *mls/evict* décrites précédemment (chapitre 2, section 2.6). La politique de gestion des niveaux de caches est non-inclusive. La variabilité du temps d'accès à la mémoire principale et aux caches partagés due aux contentions se produisant au niveau du bus mémoire est considérée connue et bornée en uti-

lisant par exemple une méthode d'accès multiple à répartition dans le temps (en anglais : TDMA pour *Time division Multiple Access*) comme proposée dans [90] ou d'autres politiques prévisibles d'arbitrage de bus [73]. Les méthodes proposées dans cette partie se focalisent sur les caches d'instructions, nous considérons donc que les caches partagés ne contiennent pas de données. De ce fait, les architectures considérées disposent soit de caches partagés séparés entre instructions et données soit d'un mécanisme permettant de partitionner les caches en A-voies d'instructions et B-voies de données au démarrage du système dans le cas de caches unifiés. La figure 3.1 illustre deux mises en œuvre possibles d'architectures multi-cœurs supportées par notre analyse.

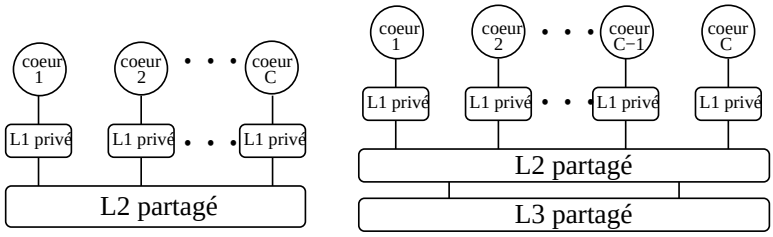


FIG. 3.1: Deux exemples d'architectures multi-cœurs supportées par nos analyses.

Concernant l'ordonnancement des tâches, nous considérons que les tâches ne peuvent pas migrer pendant une instance d'exécution. Les migrations sont permises uniquement entre deux instances d'exécution. Nous ne posons volontairement pas d'autres hypothèses sur l'ordonnancement, impliquant que n'importe quelle partie d'une tâche interférente peut être exécutée simultanément avec la tâche analysée et donc polluer les caches partagés. Cette considération permet de garder l'estimation du pire temps d'exécution indépendante des tests d'ordonnançabilité ce qui est généralement supposé lors de la validation temporelle des systèmes temps-réel. Finalement, les tâches sont considérées indépendantes (pas de synchronisations entre les tâches) mais peuvent tout de même partager du code comme par exemple le code de bibliothèques.

3.4 Détection et prise en compte des conflits dans les caches partagés

Comparée à son exécution de façon isolée, l'exécution d'une tâche sur une architecture multi-cœurs disposant de caches partagés peut introduire des défauts de cache supplémentaires dans les caches partagés à cause des tâches interférentes exécutées simultanément sur les autres cœurs. Au niveau de l'analyse statique,

cela signifie que certains accès classifiés précédemment en *Always-Hit* ou *First-Miss* lors d'une analyse de cache uni-cœur, c'est-à-dire sans prendre en compte les interférences résultantes de l'exécution simultanée de tâches sur d'autres cœurs, peuvent être modifiés en *First-Miss* ou *Not Classified* à cause des interférences inter-tâches.

De ce fait, l'analyse statique des niveaux de caches partagés doit alors être modifiée comparativement aux analyses uni-cœur, décrites dans le chapitre 2, pour prendre en compte ces interférences, tandis que l'analyse des niveaux privés reste inchangée. La figure 3.2 illustre les modifications apportées à l'analyse de cache d'un niveau partagé pour estimer et prendre en compte les conflits. L'analyse d'un niveau de cache partagé estime le pire nombre de conflits se produisant dans chacun des ensembles de ce cache, causés par l'exécution d'autres tâches sur les autres cœurs. Suite à cette estimation, l'analyse détermine la classification de comportement de chaque accès en tenant compte de ces conflits. Ces deux étapes sont détaillées dans les deux paragraphes suivants, puis nous verrons comment prendre en compte les conflits occasionnés par du code partagé.

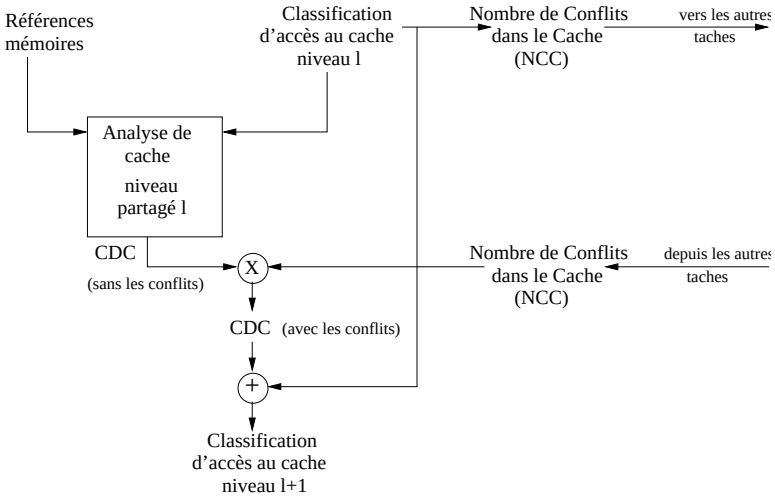


FIG. 3.2: Concept de l'analyse statique de hiérarchie de caches en présence de niveaux partagés. La classification d'accès au cache de chacune des tâches permet de déterminer le nombre de conflits dans un niveau partagé. Ces conflits sont ensuite pris en compte lors de la classification de comportement des accès mémoire afin de tenir compte des interférences inter-tâches se produisant dans le niveau partagé.

3.4.1 Détection des conflits dans un niveau de cache partagé

Pour chaque niveau de cache partagé ℓ, l'analyse statique de chacune des tâches interférentes permet de déterminer l'ensemble des accès pouvant se produire au niveau ℓ grâce à la classification d'accès au cache. Chaque accès pouvant être propagé au niveau ℓ ($CAC_{r,\ell} \neq N$) doit être considéré comme interférent, sans se préoccuper de quand cet accès peut se produire pour rester indépendant de l'ordonnancement. Nous calculons donc une borne supérieure sûre du nombre de lignes de cache en conflits pour chacun des ensembles de cache du niveau partagé ℓ. Ce nombre pour un ensemble de cache e est appelé nombre de conflits dans le cache ($NCC(e)$) et est calculé de la façon suivante :

$$NCC(e) = \sum_{t}^{TI} \mid \{cl \in t, (\exists r \in cl, CAC_{r,\ell} \neq N) \wedge mapped(cl, e)\} \mid$$

où TI est l'ensemble des tâches interférentes, cl est une ligne de cache, r est une référence mémoire contenue dans cl et $mapped(cl, e)$ est une fonction booléenne retournant vrai si cl est projetée dans l'ensemble e du cache et faux dans le cas contraire.

3.4.2 Prise en compte des conflits dans un niveau de cache partagé

Le nombre de conflits dans le cache est utilisé avec les états de cache abstraits fournis par les analyses de cache *Must* et *Persistence* pour déterminer la classification de comportement prenant en compte les interférences inter-tâches. Les états de cache abstraits produits par l'analyse *Must* conservent l'âge maximal de chaque ligne de cache cl contenue dans un ensemble de cache e. Prendre en compte les interférences implique que dans le pire cas, l'âge de cl doit être incrémenté par le nombre de conflits dans le cache déterminé dans l'ensemble e ($NCC(e)$). Si cette valeur est inférieure ou égale au degré d'associativité du cache, alors la ligne de cache est assurée d'être présente dans le cache sinon, la ligne de cache est considérée absente de l'état de cache abstrait de l'analyse *Must*. La figure 3.3 illustre ces deux cas pour un cache disposant d'un degré d'associativité de deux. La même procédure est appliquée pour l'analyse de *Persistence*, celle-ci conservant également l'âge maximal de chacune des lignes de cache. Inversement, pour l'analyse *May* déterminant si une ligne de cache peut être présente dans le cache, aucune modification n'est requise.

La classification de comportement ainsi obtenue est nécessairement plus pessimiste que celle obtenue sans prendre en compte les interférences, ce qui indirecte-

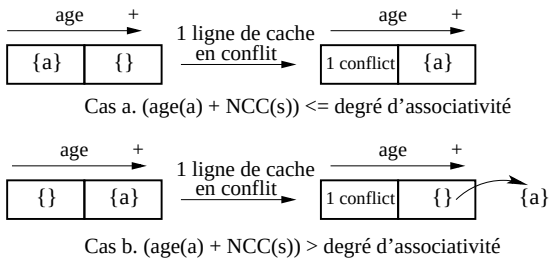


FIG. 3.3: Prise en compte des conflits dans un niveau de cache partagé ayant un degré d'associativité de deux lors de l'analyse statique. Le nombre de conflits pouvant se produire dans le cache partagé est ajouté à l'âge de la ligne de cache contenant la référence mémoire a. Si la valeur résultante est inférieure au degré d'associativité (cas a), la ligne de cache est considérée comme présente dans le cache même en présence de conflits et absente dans le cas contraire (cas b).

ment peut également modifier la classification d'accès au cache du niveau suivant. Pour garantir la sûreté de l'analyse dans le cas d'une architecture constituée de plusieurs niveaux de caches partagés, nous analysons de façon séquentielle chaque niveau de cache partagé pour l'ensemble des tâches avant d'analyser le niveau suivant. Cette approche séquentielle permet de fournir une classification d'accès au cache au niveau suivant intégrant l'effet des conflits pour la tâche analysée ainsi que pour l'ensemble des tâches interférentes, permettant alors d'estimer correctement les conflits possibles dans le niveau suivant.

3.4.3 Traitement des conflits dus au code partagé

Pour prendre en compte les effets du partage de code, comme des librairies partagées entre les tâches, nous séparons la détection des conflits en deux parties distinctes : le nombre de conflits dans le cache pour la partie privée du code ($NCC_{private}$) et le nombre de conflits dans le cache pour la partie partagée du code (NCC_{shared}) en considérant que l'ensemble des adresses du code partagé est connu et passé en paramètre de l'analyse. Cette distinction est faite car les interférences provoquées par les lignes de cache appartenant au code partagé ne sont pas toujours destructives.

Le nombre de conflits dans le cache de la partie privée du code est calculé comme précédemment mais, sans prendre en compte les adresses du code partagé ($NCC(e) \simeq NCC_{private}(e)$).

Le calcul du nombre de conflits dans le cache de la partie partagée du code pour une ligne de cache cl ($NCC_{shared}(e, cl)$) projetée dans l'ensemble e du cache est calculé en deux phases successives :

– La première étape détermine l'ensemble, noté $Shared(e)$, des lignes de cache

appartenant au code partagé et source d'interférences dans l'ensemble e du cache. En raison du partage de code, chaque ligne de cache appartenant à une librairie partagée et utilisée par au moins une tâche interférente doit être considérée comme interférente mais, uniquement une seule fois car nous considérons que l'ensemble des lignes de cache interférentes peut interférer au même instant pour être indépendant de l'ordonnancement. De ce fait, l'ensemble $Shared(e)$ est défini comme l'union, pour toutes les tâches interférentes, de l'ensemble des lignes de cache appartenant au code partagé, projetées dans l'ensemble e et pouvant accéder le niveau de cache ℓ analysé. Plus formellement, nous avons :

$$Shared(e) = \bigcup_{t}^{TI} \{cl \in t, (\exists r \in cl, CAC_{r,\ell} \neq N) \wedge shared\_code(cl) \wedge mapped(cl, e)\}$$

où $shared\_code(cl)$ est une fonction booléenne retournant vrai si la ligne de cache cl appartient à la partie du code partagé et faux dans le cas contraire.

– La seconde étape pour calculer $NCC_{shared}(e, cl)$ provient du fait que la tâche analysée peut également utiliser des lignes de cache présentes dans l'ensemble $Shared(e)$. Prenons le cas de l'analyse *Must*, à chaque point de programme, une ligne de cache cl' appartenant à l'ensemble $Shared(e)$ est considérée comme interférente avec la ligne de cache cl et donc comptabilisée dans $NCC_{shared}(e, cl)$ si et seulement si l'âge de cl' est strictement supérieur à l'âge de cl dans l'état de cache abstrait de l'analyse *Must* (ACS_{Must}) ou si cl' est absente de l'ACS_{Must}. De façon plus formelle, nous avons :

$$NCC_{shared}(e, cl) =\mid Shared(e) \setminus \{cl' \in ACS_{Must}, age(cl') \leq age(cl)\} \mid$$

Dans le cas contraire, si cl' est présente dans l'état de cache abstrait et que son âge est inférieur, cela signifie que cl' a été référencée plus récemment que cl par la tâche analysée et de ce fait, l'interférence causée par cl' est d'ores et déjà prise en compte dans l'état de cache abstrait de l'analyse *Must*.
En suivant le même raisonnement, nous obtenons pour l'analyse *Persistence* :

$$NCC_{shared}(e, cl) =\mid Shared(e) \setminus \{cl' \in ACS_{Persistence}, age(cl') \leq age(cl)\} \mid$$

Finalement, pour les deux analyses *Must* et *Persistence*, une ligne de cache cl est considérée absente de l'état de cache abstrait lors de l'analyse si la somme

des conflits avec son âge est supérieure au degré d'associativité du cache, ce qui donne formellement :

$$age(cl) + NCC_{private}(e) + NCC_{shared}(e, cl) > \text{degré d'associativité}$$

Pour l'analyse *May*, contrairement à l'analyse sans code partagé où aucune modification n'était nécessaire, il faut effectuer un post traitement sur les références au code partagé classifiées *Always-Miss* pour prendre en compte le fait que les lignes de cache contenant ces références peuvent être chargées par les tâches interférentes et donc présentes dans le cache. De ce fait, toutes références au code partagé classifiées *Always-Miss* doivent être classifiées *Not Classified*.

De plus, pour des architectures propices aux anomalies temporelles, il faut également prendre en compte le fait que de telles lignes de cache peuvent être en cours de chargement. Pour ce faire, nous ajoutons la classification *Half-Miss* (HM) à la classification de comportement :

– *Half-Miss* (HM) : La référence peut-être présente, absente ou en cours de chargement dans le cache.

Pour obtenir une classification sûre pour une architecture pouvant provoquer des anomalies temporelles, il faut modifier toutes les références au code partagé classifiées *Always-Miss* ou *Not Classified* en *Half-Miss*.

3.5 Réduction des conflits en utilisant un mécanisme de *bypass*

La sûreté de cette approche d'estimation de pire temps d'exécution pour des architectures multi-cœurs en présence de caches partagés est assurée en considérant l'ensemble des lignes de cache des tâches interférentes comme étant toujours en conflit à tout instant. Néanmoins, cette approche seule souffre, comme nous le verrons dans l'évaluation expérimentale, d'un fort pessimisme amenant rapidement l'analyse à ne détecter aucune réutilisation au niveau des caches partagés.

Pour pallier cette limitation, nous proposons une méthode visant à réduire le nombre de conflits dans les niveaux partagés. Lors d'un défaut de cache, l'opération standard consiste à récupérer la ligne de cache manquante à partir des niveaux inférieurs et à la stocker dans les niveaux supérieurs de la hiérarchie. Néanmoins, il est difficile d'affirmer que le stockage de cette ligne de cache dans les niveaux intermédiaires est réellement utile. Dans certains cas, une ligne de cache stockée dans un niveau après un défaut peut être évincée avant d'avoir été accédée de nouveau. De telles lignes de cache, appelées lignes de cache à usage unique, contribuent à un phénomène bien connu de pollution de cache [75]. Dans [75], les lignes de cache non réutilisées et donc participant à ce phénomène

de pollution ont été évaluées à 33% en moyenne dans le cache de second niveau pour les programmes de test SPEC CPU 2000.

Des méthodes basées sur un mécanisme de *bypass* [31, 33, 75, 74] ont d'ores et déjà été proposées et certaines architectures comme HP PA-RISC et Itanium dispose d'un support ISA (*Instruction Set Architecture*) permettant son utilisation. Ce mécanisme consiste à court-circuiter le stockage de certaines lignes de cache dans un niveau de cache et ainsi réduire le phénomène de pollution. Dans [75], la détection ainsi que le *bypass* des lignes de cache à usage unique sont dynamiques et mis en œuvre par matériel. Dans [33], une méthode de profilage ainsi qu'une solution dynamique pour détecter les lignes de cache à court-circuiter est définie. Néanmoins, ces travaux ont pour but de réduire le temps d'exécution moyen sans se préoccuper du pire cas.

L'utilisation directe de ces méthodes, dans le cadre de l'estimation du pire temps d'exécution, serait source d'indéterminisme à cause de leur dimension dynamique. Cependant, les méthodes d'analyses statiques de caches offrent la possibilité de déterminer statiquement les lignes de cache à usage unique, du moins celles considérées comme telles lors de l'estimation du pire temps d'exécution, rendant alors ce type d'approche déterministe. En partant de cette observation, notre approche visant à réduire le nombre de conflits dans les niveaux partagés consiste d'une part à déterminer *statiquement* les lignes de cache à usage unique pour *court-circuiter* leur stockage dans les niveaux de caches partagés lors de l'exécution puis d'autre part, à intégrer ce mécanisme dans l'approche précédente d'estimation de conflits lors de l'estimation du pire temps d'exécution.

3.5.1 Identification statique des lignes de cache à usage unique

Pour un niveau de cache partagé ℓ, une ligne de cache est déterminée statiquement comme étant à usage unique si et seulement si quel que soit le contexte d'exécution, cette ligne de cache n'est pas source de réutilisation dans le niveau ℓ lorsqu'elle est accédée. En d'autres termes, elle est toujours évincée avant d'être accédée de nouveau. L'ensemble des lignes de cache à usage unique est déterminé statiquement pour chacune des tâches en les considérant en isolation, c'est-à-dire sans prendre en compte les conflits provoqués par les tâches interférentes, en se basant sur la classification de comportement ainsi que la classification d'accès au cache de chaque référence mémoire résultant de l'analyse de cache du niveau ℓ (partie du haut de la figure 3.4). Le fait de déterminer cet ensemble en considérant les tâches en isolation permet de s'assurer que l'ajout du mécanisme de *bypass* pour les lignes de cache déterminées statiquement comme étant à usage unique n'augmente pas la valeur estimée du pire temps d'exécution, celles-ci étant d'ores et déjà considérées absentes lors de l'estimation.

Formellement, nous définissons la fonction booléenne f_{SSU} (table 3.1) qui retourne vrai si l'accès à une référence r dans un contexte d'exécution c[1] n'est pas garanti d'être un cas de réutilisation et faux dans le cas contraire. Nous pouvons remarquer que l'accès à une référence classifiée *First-Miss* et disposant de la classification d'accès *Uncertain-Never* n'est pas un cas de réutilisation par définition de la classification *Uncertain-Never*, la fonction f_{SSU} retourne donc vrai dans ce cas.

$CAC_{r,\ell,c}$ \ $CDC_{r,\ell,c}$	AM	AH	FM	NC
A	true	false	false	true
N	true	true	true	true
U-N	true	false	true	true
U	true	false	false	true

TAB. 3.1: $f_{SSU}(CAC_{r,\ell,c}, CDC_{r,\ell,c})$ du niveau ℓ

Finalement, une ligne de cache *cl* est détectée par l'analyse comme étant à usage unique si l'ensemble des accès à cette ligne n'est pas source de réutilisation ce qui donne formellement :

$$\bigwedge_{c \in contextes} \bigwedge_{r \in cl} f_{SSU}(CAC_{r,\ell,c}, CDC_{r,\ell,c})$$

avec *contextes* représentant l'ensemble des contextes d'exécution.

3.5.2 Prise en compte du mécanisme de *bypass* lors de l'analyse de cache des niveaux partagés

La partie du bas de la figure 3.4 illustre l'intégration du mécanisme de *bypass* des lignes de cache à usage unique lors de l'analyse d'un niveau de cache partagé. L'information indiquant qu'une ligne de cache est détectée comme étant à usage unique est véhiculée par la classification d'accès au cache. Concrètement, nous modifions cette classification en *Bypass* pour l'ensemble des références mémoire contenues dans une telle ligne, permettant ainsi d'indiquer à l'analyse que la ligne accédée ne sera pas stockée dans le cache partagé lors de l'exécution. Vis-à-vis de l'analyse, un accès classifié *Bypass* est sémantiquement équivalent à un accès classifié *Never* et ne modifie donc pas l'état de cache abstrait. Pour l'estimation des conflits, le calcul s'effectue comme précédemment en ne considérant pas les accès classifiés *Bypass* comme interférents dans le cache partagé. Concernant la classification de comportement des accès classifiés *Bypass*, celle ci est mise à

[1] La notion de contexte jusqu'ici implicite pour la classification de comportement et la classification d'accès au cache devient explicite pour la détection des lignes de cache à usage unique.

Always-Miss lors de l'analyse. De ce fait, aucune modification interne de l'analyse de cache n'est nécessaire pour prendre en compte le mécanisme de *bypass*.

Finalement, pour garantir la sûreté de l'estimation du pire temps d'exécution lors de l'analyse d'une hiérarchie composée de plusieurs niveaux de caches partagés, la classification d'accès au cache de niveau ℓ des lignes de cache à usage unique avant sa modification en *Bypass* doit être directement transmise au niveau $\ell + 1$ puisque le mécanisme de *bypass* impacte uniquement le contenu du niveau ℓ.

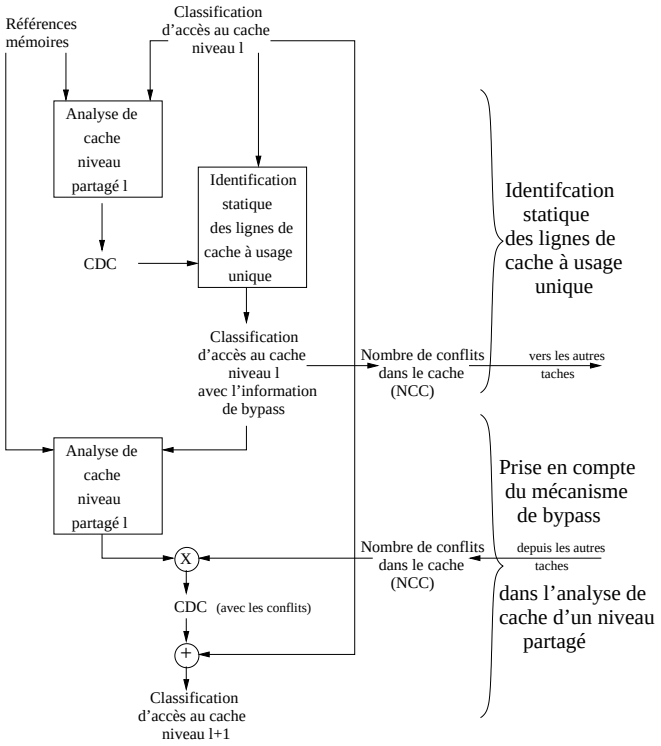


FIG. 3.4: Concept de l'analyse statique de hiérarchie de caches couplée au mécanisme de *bypass*. La première partie de l'analyse (haut de la figure) identifie les lignes de cache considérées comme étant à usage unique lors de l'estimation du pire temps d'exécution de la tâche en isolation. La seconde partie (bas de la figure) intègre la notion du mécanisme de *bypass* servant à réduire les conflits lors de l'analyse d'un niveau de cache partagé.

3.5.3 Support matériel pour utiliser notre stratégie de *bypass*

Afin d'utiliser notre stratégie de *bypass*, un support matériel est requis au sein de l'architecture. Une approche simple pour le mettre en œuvre consiste à

utiliser la méthode proposée dans [76] où les instructions disposent d'un bit dédié pour contrôler leur mise en cache. Après l'estimation des lignes de cache à usage unique, ce bit peut être renseigné pendant la phase de compilation sans aucune modification du plan mémoire. Pour des niveaux multiples de caches partagés, cette solution nécessite n bits où n représente le nombre de caches partagés (en pratique un ou deux bits pour les architectures standard). D'autres mises en œuvre plus complexes ont été suggérées dans [64] comme la distinction entre les instructions à mettre en cache et celles à ne pas mettre en se basant sur leurs adresses, ou encore d'activer/désactiver dynamiquement la mise en cache des instructions par une instruction spécifique. Néanmoins, ces mises en œuvre nécessitent un support plus important du compilateur à cause des modifications du plan mémoire qu'elles engendrent.

3.6 Expérimentations

3.6.1 Conditions expérimentales

Programme de test

Les expérimentations sont menées sur huit programmes de test issus des *WCET benchmarks* maintenus par le groupe de recherche en WCET de Mälardalen [62]. Le tableau 3.2 donne les caractéristiques de chacun des programmes de test. Les programmes de test sont en partie différents de ceux utilisés dans le chapitre précédent afin d'illustrer tous les comportements que nous avons observé avec notre approche basé sur un mécanisme de *bypass*.

Nom	**Description**	**Taille du code** (octets)
crc	Contrôle de redondance cyclique	1432
qurt	Calcul des racines d'équations quadratiques	1928
lms	Amélioration adaptatif de signal	2828
fdct	Transformée en cosinus discrète	3468
fft	Transformée de Fourier rapide	3536
minver	Inversion d'une matrice 3x3 composée de nombre flottants	4408
adpcm	Encodage de la voix	7740
statemate	Code généré automatiquement par STARC (STAtechart Real-time-Code generator)	8900

Tab. 3.2: Caractéristiques des programmes de test

Analyse de caches et estimation du pire temps d'exécution

Les conditions expérimentales vis-à-vis de l'analyse de caches et de l'estimation du pire temps d'exécution sont identiques au chapitre précédent. Pour rappel :

- le code binaire est compilé avec gcc 4.1 sans optimisation pour du MIPS R2000/R3000 [66] ;
- l'estimation du pire temps d'exécution est calculé par le logiciel HEPTANE [24] en utilisant la méthode IPET ;
- l'analyse est contextuelle ;
- l'état initial du cache est considéré vide (absence d'anomalies temporelles) ;
- chaque niveau de cache dispose d'une politique de remplacement de cache LRU.

Configuration de cache

Les expérimentations sont menées en utilisant une hiérarchie de cache non-inclusive composée de deux niveaux mettant en œuvre la politique de remplacement de cache LRU. Le premier niveau de cache (L1) est privé et dispose d'un degré d'associativité de 4 pour une taille de 1KB. Le second niveau (L2) est quand à lui partagé entre les différents cœurs et dispose d'une taille de 4KB avec un degré d'associativité de 8. Les tailles des lignes de caches sont de 32B pour les deux niveaux de la hiérarchie. Des latences de 1 cycle pour le L1, de 10 cycles pour le L2 et de 100 cycles pour la mémoire principale sont utilisées.

Métriques

Dans cette partie, nous utilisons deux métriques pour évaluer les performances de l'analyse de hiérarchies de caches en présence de cache partagé. La première métrique est le taux de succès (noté HR_ℓ) dans les caches L1 et L2 le long du pire chemin d'exécution (en anglais WCEP pour *Worst Case Execution Path*). Le pire chemin d'exécution est obtenu en utilisant les fréquences des blocs de base retournée par la méthode d'énumération implicite des chemins. La seconde métrique utilisée est le nombre de cycles par instruction causés par les défauts de cache L2 (noté $NCPI$[2]) le long du pire chemin d'exécution. Cette seconde métrique est complémentaire de la première et permet de mieux quantifier l'impact en terme de cycles de notre approche de *bypass*. Nous ne comparons pas les résultats obtenus par analyse avec des mesures en raison de la difficulté à se placer dans les pires conditions d'exécution.

[2] $NCPI = \frac{\#L2\ miss * memory\ latency}{\#instruction\ along\ the\ WCEP}$

3.6.2 Résultats expérimentaux

Impact du mécanisme de *bypass* pour un système sans interférences au niveau du cache partagé

Nous regardons dans un premier temps l'impact du mécanisme de *bypass* pour un système sans interférences au niveau du cache partagé. Pour ce faire, nous comparons les résultats de l'analyse obtenus pour chaque programme de test avec et sans le mécanisme de *bypass* en considérant l'exécution de la tâche sur un cœur et sans tâches interférentes. Le tableau 3.3 présente pour chaque programme de test (colonne 1) le taux de succès dans le cache L1 (colonne 2) le taux de succès dans le cache L2 ainsi que le nombre de cycles par instruction causés par les défauts de cache L2 sans (colonne 3) et avec (colonne 4) le mécanisme de *bypass*. Le taux de lignes de cache non chargées dans le cache L2 par le mécanisme de bypass (noté *bypass ratio*, colonne 5) est également fourni afin d'indiquer la réduction possible de la pollution dans le cache L2. Plus ce taux est élevé, plus la réduction possible au niveau des interférences inter-tâches est grande.

Bench.	**HR$_{L1}$**	**L2 sans bypass** HR$_{L2}$ (NCPI)	**L2 bypass** HR$_{L2}$ (NCPI)	**ratio du bypass**
minver	93.99%	39.76% (3.62)	39.76% (3.62)	94.92%
adpcm	89.74%	33.60% (6.81)	33.96% (6.77)	88.02%
fdct	87.25%	84.03% (2.04)	84.03% (2.04)	5.5%
statemate	83.40%	0.72% (16.49)	1.21% (16.40)	98,92%
fft	88.76%	1.97% (11.02)	12.50% (9.83)	92.79%
crc	93.10%	98.97% (0.07)	98.97% (0.07)	88.89%
lms	87.24%	0.61% (12.68)	0.61% (12.68)	94.38%
qurt	93.57%	12.56% (5.62)	12.56% (5.62)	98.36%

TAB. 3.3: Impact du mécanisme de *bypass* du cache L2 sur l'estimation du pire temps d'exécution pour un système sans interférence.

L'utilisation du mécanisme de *bypass* permet d'obtenir un taux de succès dans le cache L2 toujours supérieur ou égal à celui obtenu sans ce mécanisme. Pour trois des programmes de test (*statemate*,*adpcm* et *fft*), ce taux est strictement supérieur lorsque le mécanisme de *bypass* est utilisé. Pour ces programmes, le fait de ne pas stocker les lignes de cache détectées comme étant à usage unique permet de détecter de la réutilisation supplémentaire et donc plus de succès de cache pour les autres lignes de cache grâce à la réduction du nombre de conflits intra-tâche. Dans le meilleur des cas (*fft*), HR_{L2} est multiplié par un facteur 6 réduisant alors de façon significative l'estimation du pire temps d'exécution (plus d'un cycle par instruction). Pour les deux autres, l'amélioration causée par cet effet secondaire est moins significative.

En terme de nombre de lignes de cache non stockées dans le cache L2, le

pourcentage est élevé pour tous les programmes de test sauf *fdct*. Ce pourcentage élevé montre que la pollution du cache L2 peut être réduite de façon importante par le mécanisme de *bypass* et donc réduire de façon significative les interférences inter-tâches. Pour *fdct*, le pourcentage faible s'explique par la structure du code constituée de deux boucles ne tenant pas dans le cache L1 tandis qu'elles sont contenues intégralement dans le cache L2.

Analyse de cache partagé avec et sans mécanisme de *bypass*

Nous nous intéressons dans cette partie à l'estimation du pire temps d'exécution d'une tâche s'exécutant sur un cœur en concurrence au niveau du cache L2 avec une autre tâche exécutée sur l'autre cœur, dans un contexte où il n'y a pas de partage de code entre les tâches. La tâche en interférence au niveau du cache L2 avec la tâche analysée est l'une des huit tâches présentées dans le tableau 3.3.

Les résultats sont présentés dans le tableau 3.4. Pour chaque programme de test, les résultats sans (première ligne) et avec (seconde ligne) le mécanisme de *bypass* sont fournis. Pour chaque configuration, le taux de succès dans le cache, ainsi que le nombre de cycles par instruction causé par les défauts de cache L2 sont donnés : (*i*) pour la tâche analysée considérée sans aucune tâche en interférence, (*ii*) en prenant la moyenne obtenue à partir des résultats de chacune des tâches en interférence, (*iii*) pour la tâche analysée considérée en interférence avec la tâche provoquant le plus d'interférence.

En regardant la moyenne obtenue à partir des résultats de chacune des tâches considérées comme interférentes nous observons que l'augmentation du taux de succès au niveau du cache L2 en comparaison d'un système ne disposant pas de mécanisme de *bypass* est significative tandis que l'impact sur le nombre de cycles par instruction causé par les défauts de cache L2 est relativement minime pour une majorité des programmes de test.

Dans le pire cas par contre (colonne pire tâche interférente), l'analyse simple sans le mécanisme de *bypass* amène toujours à un résultat où le taux de succès dans le cache L2 est de 0% et dans ce cas le nombre de cycles par instruction causé par les défauts de cache L2 est maximal. Ce résultat ne se produit pas toujours avec la même tâche en interférence suivant la tâche analysée. De plus, plusieurs tâches interférentes amènent à ce résultat. En d'autres termes, l'estimation du pire temps d'exécution en absence du mécanisme de *bypass* montre le pessimisme des méthodes, comme celle proposée précédemment, du fait de considérer l'ensemble des interférences inter-tâches sans mécanisme pour les réduire.

Intéressons nous maintenant aux résultats obtenus lorsque le mécanisme de *bypass* est utilisé. Si nous comparons le taux de succès dans le cache L2 lorsque la tâche analysée est en concurrence avec la pire tâche interférente et ce même taux de succès lorsque la tâche est en isolation, nous observons que pour six pro-

tâche	**bypass**	**tâche en isolation** HR_{L2} (NCPI)	**moyenne des tâches interférentes** HR_{L2} (NCPI)	**pire tâche interférente** HR_{L2} (NCPI)
minver	non	39.76% (3.62)	24.85% (4.51)	0% (6.01)
	oui	39.76% (3.62)	39.76% (3.62)	39.76% (3.62)
adpcm	non	33.60% (6.81))	20.60% (8.14)	0% (10.26)
	oui	33.96% (6.77)	33.76% (6.80)	32.90% (6.88)
fdct	non	84.03% (2.04)	24.88% (9.58)	0% (12.75)
	oui	84.03% (2.04)	73.32% (3.40)	7.34% (11.81)
statemate	non	0.72% (16.49)	0.19% (16.57)	0% (16.60)
	oui	1.21% (16.40)	1.21% (16.40)	1.21% (16.40)
fft	non	1.97% (11.02)	1.23% (11.10)	0% (11.24)
	oui	12.50% (9.83)	12.50% (9.83)	12.50% (9.83)
crc	non	98.97% (0.07)	61.65% (2.63)	0% (6.90)
	oui	98.97% (0.07)	98.97% (0.07)	98.97% (0.07)
lms	non	0.61% (12.68)	0.38% (12.70)	0% (12.75)
	oui	0.61% (12.68)	0.61% (12.68)	0.61% (12.68)
qurt	non	12.56% (5.62)	7.85% (5.92)	0% (6.43)
	oui	12.56% (5.62)	12.56% (5.62)	12.56% (5.62)

TAB. 3.4: Valeur estimée du pire taux de succès de cache dans le cache L2 ainsi que le NCPI de la tâche analysée avec et sans mécanisme de *bypass* en présence d'une tâche interférente.

grammes de test les résultats sont identiques. Ils sont très proches pour *adpcm* (1.04%). En revanche pour *fdct*, la différence est significative (76.69%). Ce résultat est obtenu en mettant *fdct* en interférence avec elle même (pour rappel, nous ne considérons pas le code partagé dans cette partie) et nous avons observé précédemment que cette tâche disposait d'une faible proportion de lignes de cache à usage unique. Lorsque *fdct* n'est plus en conflit avec elle même, la pire des tâches interférentes restante amène à un taux de succès dans le cache L2 de 75.06% et la différence avec l'analyse de la tâche en isolation devient inférieur à 9%. En terme de nombre de cycles par instruction causé par les défauts de cache L2, l'amélioration moyenne comparativement à un système sans mécanisme de *bypass* est de 2.1 cycles par instruction et atteint au plus 6.83 cycles pour *crc*.

L'utilisation du mécanisme de *bypass* permet donc en général de réduire de façon considérable les interférences inter-tâches et ainsi obtenir une estimation du pire temps d'exécution proche de celle obtenue lorsque la tâche est considérée en isolation sauf lorsque la tâche en conflit consomme une part importante de la capacité du L2.

Passage à l'échelle

Nous proposons d'étudier le passage à l'échelle de notre approche en considérant toutes les tâches comme interférentes, chacune s'exécutant sur un cœur. Avec cette configuration, la taille totale des tâches considérées est approximativement huit fois plus grande que la capacité du cache L2. Les résultats sont présentés dans le tableau 3.5. La tâche *fdct* qui utilise une grande proportion de la capacité du cache L2 même avec le mécanisme de *bypass* n'est pas considérée comme une tâche interférente dans la colonne 3 et est considérée comme une tâche interférente dans la colonne 4. Aucune comparaison n'est faite avec les résultats de l'analyse sans mécanisme de *bypass* car il n'y a pas de succès de cache dans le L2 en présence d'une seule tâche interférente.

Bench.	**sans interférence**	**toutes les tâches en interférences**	
		sans fdct	avec fdct
	HR_{L2} (NCPI)	HR_{L2} (NCPI)	HR_{L2} (NCPI)
minver	39.76% (3.62)	38.55% (3.69)	0% (6.01)
adpcm	33.96% (6.77)	21.97% (6.83)	15.59% (8.66)
fdct	84.03% (2.04)	66.08% (4.32)	1.63% (12.54)
statemate	1.21% (16.40)	1.16% (16.41)	0% (16.60)
fft	12.50% (9.83)	6.82% (10.47)	0.88% (11.14)
crc	98.97% (0.07)	98.97% (0.07)	0% (6.90)
lms	0.61% (12.68)	0.61% (12.68)	0% (12.75)
qurt	12.56% (5.62)	12.56% (5.62)	0% (6.43)

TAB. 3.5: Valeur estimée du pire taux de succès de cache dans le cache L2 ainsi que le NCPI de la tâche analysée avec le mécanisme de *bypass* en présence de 7 et 8 tâches interférentes.

Lorsque la tâche *fdct* ne fait pas partie des tâches interférentes, la réduction du taux de succès dans le cache L2 est minime pour une majorité des programmes de test (*minver*, *statemate*, *crc*, *lms* et *qurt*) en comparaison avec le taux obtenu lorsque la tâche est analysée en isolation. Pour les autres, la réduction est plus importante cependant, nous observons tout de même des succès dans le cache L2. Nous observons une légère augmentation du nombre de cycles par instruction causé par les défauts de cache L2. En moyenne nous obtenons une augmentation de 0.38 cycle par rapport aux résultats de l'analyse considérant les tâches en isolation et dans le pire cas une augmentation de 2.28 cycles pour *fdct*. Par rapport à une analyse prenant en compte les interférences mais sans mécanisme de *bypass* l'amélioration est de 2.1 cycles en moyenne et dans le meilleur des cas l'amélioration est de 8.43 cycles pour *fdct*. Ces résultats montrent que notre approche de *bypass* permet de réduire efficacement les conflits inter-tâches et de fait l'impact de celles-ci sur l'estimation du pire temps d'exécution.

Lorsque la tâche *fdct* fait partie des tâches interférentes, la diminution du

taux de succès dans le cache L2 est significative en raison de la consommation importante de la capacité du L2 par cette tâche. Avec ce type de tâches, l'approche utilisant le mécanisme de *bypass* n'est pas suffisante pour réduire l'impact des interférences inter-tâches dans les niveaux de cache partagés. Dans ce cas, nous suggérons d'utiliser par exemple une approche de partitionnement de cache afin d'isoler les tâches consommant une part importante de la capacité des caches partagés afin de conserver les performances des autres tâches.

Prise en compte du code partagé

Nous évaluons maintenant l'impact de la prise en compte du code partagé lors de l'estimation du pire temps d'exécution. Les résultats sont présentés dans le tableau 3.6. En raison de la difficulté à trouver différents degrés de code partagé dans les applications temps-réel, l'évaluation est réalisée sur une tâche unique exécutée sur deux ou trois cœurs en considérant un certain pourcentage du code comme étant partagé. L'expérimentation est réalisée en considérant la tâche *fdct* celle-ci étant la tâche la plus consommatrice au niveau du cache L2 même en présence du mécanisme de *bypass*. La quantité de code partagé entre les instances de cette tâche varie entre 0% (i.e. pas de code partagé) et 100% (i.e. l'intégralité du code est partagé). Un pourcentage de x indique que les premiers $\frac{x*taille\_code}{100}$ octets du code sont partagés entre les instances.

% du code partagé	**1 instance en interférence**		**2 instances en interférences**	
	sans bypass NCPI	**bypass** NCPI	**sans bypass** NCPI	**bypass** NCPI
0 %	12.75	11.81	12.75	12.75
10 %	12.54	11.50	12.75	12.75
20 %	10.46	9.01	12.75	12.75
30 %	8.28	7.76	12.75	12.54
40 %	8.07	7.76	12.75	12.54
50 %	8.07	7.76	12.75	12.54
60 %	7.45	5.78	11.92	10.77
70 %	7.13	5.36	11.92	9.94
80 %	3.60	3.15	9.53	6.30
90 %	2.04	2.04	5.68	5.36
100 %	2.04	2.04	2.04	2.04

Tab. 3.6: Estimation du NCPI pour fdct avec partage de code, avec et sans le mécanisme de *bypass*, avec une ou deux instances de fdct en interférences.

Les résultats montrent que la prise en compte du code partagé permet de raffiner l'estimation du pire temps d'exécution comparativement à une analyse ne le prenant pas en compte. Nous observons également que le mécanisme de *bypass* conserve tout sont intérêt en réduisant les conflits sur les parties de code privé.

3.7 Conclusion

Dans ce chapitre, nous venons de développer la méthode publiée dans [38] décrivant une analyse statique de contenu des caches d'instructions partagés pour des processeurs multi-cœurs couplée avec un mécanisme de *bypass* pour réduire les conflits inter-tâches dans les niveaux partagés. Nous avons dans un premier temps mis en évidence les problèmes de sûreté pouvant se produire avec l'analyse existante [110]. Ensuite nous avons proposé une approche sûre permettant d'estimer le pire temps d'exécution de tâches s'exécutant sur des architectures multi-cœurs disposant de mémoire cache partagée. Cette approche considère l'ensemble des conflits possibles comme étant toujours en interférence avec la tâche analysée. Cette modélisation à également été proposée en parallèle dans [52] et présentée lors de la même conférence que nos travaux [38]. La différence principale entre notre approche et [52] pour la prise en compte des conflits se situe au niveau de la prise en compte du code partagé lors de l'analyse, qui est absente dans [52].

Comme nous l'avons vu lors des expérimentations, cette méthode utilisée seule amène rapidement l'analyse à ne détecter aucune réutilisation dans les niveaux de cache partagé ce qui peut ajouter un fort pessimisme à l'estimation du pire temps d'exécution. Nous avons proposé de coupler cette analyse avec un mécanisme de *bypass* afin de réduire les interférences inter-tâches dans les niveaux partagés. L'idée consiste à déterminer, lors de la phase de compilation, l'ensemble des lignes de cache à usage unique, afin de ne pas les stocker dans les niveaux partagés lors de l'exécution et ainsi réduire considérablement le nombre de conflits inter-tâches. Les expérimentations montrent que cette approche permet de capturer de la réutilisation dans les niveaux partagés même lorsque la taille totale des tâches s'exécutant sur le système est huit fois supérieure à la capacité du cache partagé de second niveau.

En réponse à la seconde question de recherche de ce document, nous avons montré, dans ce chapitre, la faisabilité d'analyser de façon sûre des hiérarchies de caches disposant de niveaux partagés. Concernant la précision, l'estimation du pire temps d'exécution est exploitable pour des tâches ayant une consommation raisonnable de la capacité des niveaux partagés. Pour les tâches plus gourmandes, nous suggérons d'utiliser un cloisonnement au niveau des caches partagés afin d'éliminer les interférences provoquées par ce type de tâche.

Nous avons récemment étendu cette solution au cas des caches de données partagés dans [50]. Plusieurs stratégies de *bypass* sont proposées pour mieux gérer les différents types d'accès aux données comme le *bypass* systématique de tous les accès indéterministes ceux-ci étant généralement considérés par l'analyse comme produisant uniquement des défauts de cache, le *bypass* des données non réutilisées ou encore la réduction des données stockées dans le cache à un certain pourcentage de la capacité du cache.

D'autres approches ont été proposées en parallèle de nos travaux de thèse dans [52, 99] pour traiter la problématique des caches d'instructions partagés. La méthode définie dans [52] consiste à coupler l'estimation des conflits dans les niveaux de cache partagé avec l'ordonnancement des tâches afin de limiter le nombre de tâches en conflits permettant ainsi de capturer de la réutilisation dans les niveaux partagés lors de l'analyse. Cette approche propose un processus itératif qui estime les conflits dans les niveaux partagés en fonction de l'ordonnancement sélectionné puis vérifie le respect des contraintes temporelles de chacune des tâches en calculant leur pire temps de réponse. Le processus se termine lorsque l'ordonnancement obtenu permet de garantir le respect des échéances de chacune des tâches. Cette méthode est différente de notre approche dans le sens où nous avons choisi d'être indépendant de l'ordonnancement pour offrir un plus grand degré de liberté au système mais en imposant en contre partie la présence d'un mécanisme de *bypass* au niveau de l'architecture. Néanmoins, ces deux approches essaient de réduire les conflits dans les niveaux de cache partagé et peuvent tout à fait être utilisées conjointement pour raffiner l'estimation du pire temps d'exécution de chacune des tâches.

Dans [99], une approche totalement différente a été proposée pour traiter la problématique des niveaux de cache partagés. Différentes stratégies de partitionnement de cache (par tâche ou par cœur) cumulées avec du gel de cache statique ou dynamique sont explorées afin d'une part de s'abstraire des interférences inter-tâches et d'autre part d'obtenir une connaissance précise du contenu du cache lors de l'estimation du pire temps d'exécution. Cette méthode est néanmoins plus restrictive vis-à-vis de l'utilisation du cache par chacune des tâches en réduisant la capacité disponible. Une étude comparative entre cette approche et la notre permettrait de déterminer quelle stratégie est la plus adaptée en fonction des caractéristiques des tâches.

Une piste intéressante à explorer serait d'étendre les stratégies de *bypass* aux niveaux des caches privés. En effet, nous avons pu observer comme effet de bord que le *bypass* des lignes de cache à usage unique peut, dans certains cas, améliorer l'estimation du pire temps d'exécution lorsque la tâche est en isolation en réduisant les conflits intra-tâche. Définir des stratégies de *bypass* visant explicitement à réduire ce type de conflits serait un nouveau type d'optimisation orientée pire temps d'exécution.

Finalement, les méthodes d'analyse statique visant à estimer le pire temps d'exécution de tâches exécutées sur des architectures multi-cœurs disposant de caches partagés ainsi que notre approche basée sur le mécanisme de *bypass* sont à l'heure actuelle définies uniquement pour des hiérarchies de caches disposant d'une politique de gestion non-inclusive. Étendre ces méthodes pour prendre en compte d'autres politiques de gestion, notamment la politique de gestion exclusive, permettrait d'être plus représentatif des architectures actuelles et permet-

trait d'effectuer une étude comparative entre les différentes politiques de gestion vis-à-vis du pire cas pour les architectures multi-cœurs.

Conclusion

Dans ce document, nous nous sommes intéressés à la validation des contraintes temporelles nécessaire pour garantir le bon fonctionnement des systèmes temps-réel. Plus précisément, nous nous sommes focalisés sur l'estimation du pire temps d'exécution des applications temps-réel exécutées sur des processeurs multi-cœurs. Après avoir présenté les principes de base ainsi que les méthodes utilisées pour effectuer l'estimation du pire temps d'exécution, nous nous sommes intéressés plus particulièrement aux mémoires cache présentes dans tous les processeurs actuels pour réduire les latences d'accès à la mémoire. Un tour d'horizon des méthodes existantes pour estimer le comportement temporel pire cas des mémoires cache a révélé que ces dernières considèrent dans la grande majorité des cas un seul niveau de mémoire cache tandis que les processeurs actuels disposent généralement d'une hiérarchie composée de plusieurs niveaux de mémoire cache.

Dans la première contribution de ce document, nous avons proposé une méthode d'analyse statique permettant d'étendre les analyses simple niveau existantes afin de déterminer le comportement temporel pire cas de l'ensemble des niveaux de caches composant une hiérarchie. La sûreté de notre approche repose sur une modélisation du filtrage des accès mémoire effectué par chaque niveau de la hiérarchie en introduisant notamment une nouvelle classification appelée classification d'accès au cache. Cette modélisation nous permet d'analyser les politiques de gestion non-inclusive, inclusive et exclusive ainsi que les politiques de remplacement de cache les plus courantes. Les résultats expérimentaux montrent que notre analyse permet de raffiner l'estimation du temps d'exécution pire cas en détectant la localité des applications capturée dans les niveaux inférieurs de la hiérarchie lors de l'exécution.

Cette méthode considère toutefois l'application analysée en isolation lors de l'analyse, excluant ainsi les effets des interférences provoquées par les autres applications s'exécutant sur les autres cœurs de l'architecture. Notre deuxième contribution ajoute à cette analyse la notion d'interférence nécessaire pour déterminer le pire temps d'exécution des applications exécutées sur les processeurs multi-cœurs en prenant en compte les conflits pouvant se produire dans les niveaux de cache partagés. Afin de réduire le nombre d'interférences, nous avons défini conjointement une méthode appliquée lors de la phase de compilation, qui ex-

ploite un mécanisme de *bypass* permettant d'éviter le stockage dans les niveaux partagées des lignes de cache connues statiquement comme étant à usage unique. L'étude expérimentale montre l'intérêt d'une telle méthode qui permet la détection de réutilisation lors de l'analyse même lorsque la taille des applications est largement supérieure à la capacité du cache partagé.

Problèmes ouverts et perspectives

Nous avons présenté des méthodes permettant l'analyse du comportement temporel pire cas du contenu des hiérarchies de mémoires cache des architectures mono-cœur et multi-cœurs. Nous avons discuté certaines extensions à la fin de chaque contribution comme l'extension de nos méthodes aux hiérarchies de caches de données ainsi qu'à d'autres types de politique de gestion de hiérarchie. L'utilisation des architectures multi-cœurs dans le cadre des systèmes temps-réel est un problème de recherche récent qui appelle à de nombreux défis.

Utilisation efficace des mémoires cache partagées

La problématique des caches partagés dans les architectures multi-cœurs est relativement récente en ce qui concerne les systèmes temps-réel. Peu de méthodes ont été proposées à ce jour pour estimer le pire temps d'exécution de tâches exécutées sur de tel système. Certaines recommandations ont été proposées dans [107, 27] afin d'exploiter les architectures multi-cœurs dans les systèmes temps-réel. Concernant les caches, ils proposent de les limiter à des caches privés pour éviter la problématique du partage de ressource. Néanmoins, la plupart des processeurs multi-cœurs actuels disposent de niveau de cache partagé. Afin d'exploiter les caches partagés des architectures multi-cœurs dans le cadre des systèmes temps-réel, une connaissance précise du contenu et du comportement des caches partagés est requise afin de les exploiter pleinement. Les méthodes existantes se sont focalisées soit sur du partitionnement et du gel de contenu de cache afin d'écarter totalement les interférences soit sur une estimation gros grain des interférences couplée avec une méthode visant à réduire les interférences. Ces approches permettent de détecter une certaine réutilisation au niveau des caches partagés. Néanmoins, une modélisation plus fine des interférences permettrait d'obtenir une estimation plus précise du pire temps d'exécution. Une piste intéressante pour améliorer la prévisibilité du comportement des niveaux partagés et donc l'estimation des interférences consisterait à définir des politiques de remplacement de cache plus spécifiques pour les niveaux partagés. Par exemple, certaines zones du cache pourraient être assignées à des cœurs en priorité, comme proposé dans [65] pour améliorer le temps moyen d'exécution. Ce type de solutions permettrait d'améliorer la prévisibilité des niveaux de cache partagés en

garantissant que certaines zones ne seraient pas affectées par des interférences inter-tâches. D'autres types de mémoire partagée pourraient également être envisagés comme les mémoires sur puce (en anglais : *scratchpad*) en définissant des méthodes permettant de sélectionner le contenu à stocker comme par exemple les données partagées entre les différentes tâches du système.

Estimation du pessimisme des méthodes d'analyse statique

Avoir une estimation du degré de pessimisme des méthodes d'analyse serait une information intéressante lors de la conception des systèmes pour choisir tel ou tel type d'architecture en fonction des applications à exécuter. Pour ce faire, une approche par mesure semble appropriée pour obtenir une borne inférieure du pire temps d'exécution en essayant par exemple de déterminer les pires entrelacements possibles au niveau des accès aux caches partagés. Nous avons réalisé quelques expérimentations préliminaires dans ce sens montrant que les méthodes d'estimations statiques sont relativement précises lorsque l'utilisation des caches partagés est intensive tandis que l'écart entre les valeurs mesurées et estimées peut être important lors d'une utilisation plus modérée des niveaux partagés.

Estimation pire cas des délais de migrations et de préemptions

Les processeurs multi-cœurs offrent également la possibilité de migrer une tâche d'un cœur vers un autre lors de son exécution. Le fait de migrer une tâche n'est pas sans effet sur son temps d'exécution total car le contenu des caches composant la hiérarchie est modifié lors de la migration. Une estimation du surcoût résultant des migrations devient une nécessité dès lors que l'ordonnanceur utilisé autorise les migrations. Nous avons proposé dans [41] une première approche permettant d'estimer le temps de migration pire cas d'une tâche tandis que [91] propose un mécanisme matériel pour migrer le contenu des caches entre deux instances d'exécution d'une tâche afin de masquer l'impact des migrations. Ces approches méritent d'être approfondies afin de tenir compte des différents types de gestion et des différentes structures de hiérarchies de caches.

Concernant l'estimation des délais de préemptions, de nombreuses méthodes ont été définies ces dernières années [71, 98, 97, 84, 16] tant pour les caches d'instructions que les caches de données. Néanmoins, l'ensemble de ces méthodes considère des architectures disposant d'un unique niveau de cache. Ces méthodes sont peut être directement utilisables pour analyser des hiérarchies de caches ou au contraire, elles peuvent peut-être poser des problèmes, vis-à-vis de la sûreté de l'estimation du pire temps, avec des effets similaires aux accès incertains si ils sont mal considérés lors de l'analyse. Une étude approfondie de cette problématique

apparaît essentielle pour prendre en compte les délais de préemption lors de la validation des systèmes temps-réel disposant de hiérarchies de caches.

Passage à l'échelle des méthodes d'analyse statique de cache

Les analyses statiques permettant de déterminer le comportement temporel pire cas des accès aux mémoires cache sont toutes basées sur des calculs de point-fixe. Ce type de calcul est coûteux en terme de temps de calcul dès que le programme à analyser atteint une certaine taille, en raison de la nature exponentielle de ce calcul. Pour cette raison, les outils d'analyse statique de cache se limitent à l'analyse de programmes de taille raisonnable (plusieurs centaines de kilo octets) mais ne sont pas en mesure d'analyser des programmes de taille importante.

Une solution pour pallier cette limitation est d'analyser le programme par parties puis, de combiner dans un second temps les résultats de chacune des parties pour obtenir une estimation du pire temps d'exécution du programme complet. Ce type d'approche, appelée analyse partielle [10, 8], permet d'améliorer le temps de calcul en le parallélisant et donc d'analyser des programmes de taille plus importante. Néanmoins, ce type d'approche dispose toujours d'une complexité exponentielle ce qui à terme risque de limiter cette approche. En effet, les systèmes temps-réel offrent de plus en plus de service et de fait, leur taille est en constante augmentation.

Explorer de nouvelles pistes, afin de définir des analyses statiques de cache disposant d'une complexité plus faible, aurait un intérêt certain pour analyser des programmes de tailles importantes, même si celles-ci seraient probablement légèrement moins précises comparativement aux analyses existantes.

Annexes

Preuve des bornes mls et $evict$ modifiées pour chacune des politiques de remplacement

Politique de remplacement FIFO

Les bornes originales et modifiées de mls (1) et $evict$ $(2k-1)$ sont identiques pour la politique de remplacement de cache FIFO.

Preuve : Un accès à un élément déjà présent dans le cache ne modifie pas l'état du cache lorsqu'il dispose d'une politique de remplacement FIFO. □

Politique de remplacement RANDOM

La borne mls est égale à 1 pour la politique de remplacement de cache RANDOM.
Preuve : Un élément peut être évincé par un accès à un élément distinct. □

La borne $evict$ est égale à ∞ pour la politique de remplacement de cache RANDOM.
Preuve : Un élément peut ne jamais être sélectionné pour être évincé. □

Politique de remplacement PLRU

Les bornes originale et modifiée de mls $(log_2(k)+1)$ sont identiques pour la politique de remplacement de cache PLRU.

Preuve : Après l'accès à un élément e, le chemin d'accès vers e, c'est à dire la valeur de chacun des nœuds dans l'arbre binaire, est 0...0. Pour remplacer e, tous les bits de son chemin d'accès doivent être mis à 1. Comme démontré dans [86], Il faut $log_2(k)+1$ accès distincts successifs pour réaliser ce changement car chaque accès change la valeur d'un seul bit. Dans notre cas, il faut en plus prendre en compte l'effet d'accès à un élément déjà présent dans le cache. L'accès à un élément e' déjà présent dans le cache ne peut pas changer plus d'un bit du chemin

d'accès vers e. L'accès à des éléments déjà présent dans le cache ne modifie donc pas cette borne. □

La borne modifiée de *evict* est 2 si $k = 2$ et ∞ sinon pour la politique de remplacement de cache PLRU.

Preuve :
- k=2
  Quand le degré d'associativité est de 2, le chemin d'accès est composé d'un seul bit. Après l'accès à un élément e, ce bit est à 0. L'accès suivant a un élément e' ($e \neq e'$) rechange la valeur de ce bit, indiquant que le prochain accès différent de e' et de e évincera e tandis que l'état du cache n'est pas modifié si e' est accédé de nouveau.
- k>2
  La séquence d'accès infinie [$abcdcecfcgch...$] n'évincera jamais l'élément a dans un cache disposant d'un degré d'associativité de 4. Le même type de séquence peut être construit pour des degrés d'associativités supérieurs strictement à 2.

□

Politique de remplacement MRU

Nous supposons que lors d'un défaut de cache la ligne de cache disposant de l'index le plus bas (celui le plus à gauche dans notre représentation) et dont le bit MRU est à 0, est évincée.
Les bornes originale et modifiée de *mls* (2) sont identiques pour la politique de remplacement de cache MRU.

Preuve : Le bit MRU d'un élément accédé e est toujours mis à 1. L'accès suivant à un élément différent e' peut remettre à 0 tous les bits MRU. Si e est l'élément le plus à gauche, il sera évincé lors de l'accès suivant à un élément différent tandis que si e' est accédé de nouveau, l'état du cache est inchangé. □

Les bornes originale et modifiée de *evict* ($2k-2$) sont identiques pour la politique de remplacement de cache MRU.

Preuve : A un point quelconque lors de l'exécution d'une séquence de taille arbitraire disposant de k différents accès (produisant un succès ou un défaut), tous les bits MRU sont remis à 0. Après cette remise à 0, $k-1$ MRU bits sont à 0. Une séquence additionnelle de taille arbitraire disposant de $k-1$ accès différent est suffisante pour évincer le premier élément de la séquence tandis qu'un accès à un élément déjà accédé peut contribuer à cette éviction (en changeant la valeur

d'un bit MRU à 1) ou peut laisser l'état du cache inchangé (si le bit MRU est déjà à 1). □

Publications de l'auteur

[1] D. Hardy and I. Puaut. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture*, 2010, To appear.

[2] B. Lesage, D. Hardy, and I. Puaut. Shared data cache conflicts reduction for WCET computations in multi-core architectures. In *Proceedings of the 18th Real-Time and Network Systems*, Toulouse, France, November 2010, To appear.

[3] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, Washington D.C., USA, December 2009.

[4] D. Hardy and I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *Proceedings of the 17th Real-Time and Network Systems*, pages 45–54, Paris, France, October 2009.

[5] B. Lesage, D. Hardy, and I.Puaut. WCET analysis of multi-level set-associative data caches. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[6] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th Real-Time Systems Symposium*, pages 456–466, Barcelona, Spain, December 2008.

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

[8] I. Puaut and D. Hardy. Predictable paging in real-time systems : a compiler approach. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 169–178, Pisa, Italy, July 2007.

Bibliographie

[1] N. B. H. Aissa, G. Grimaud, and D. Simplot-Ryl. A distributed and verifiable loop bounding algorithm for wcet computation on constrained embedded systems. In *Proceedings of the 14th International Conference on Real-Time and Network Systems*, 2006.

[2] N. B. H. Aissa, C. Rippert, D. Deville, and G. Grimaud. A distributed WCET computation scheme for smart card operating systems. In *4th international workshop on Worst Case Execution Time Analysis*, 2004.

[3] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *ACM-SE 42 : Proceedings of the 42nd annual Southeast regional conference*, pages 267–272, New York, NY, USA, 2004. ACM.

[4] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, 1994.

[5] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8 :284–292, 1993.

[6] *AUTOSAR : AUTomotive Open System ARchitecture.* http ://www.autosar.org.

[7] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA'88 : Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[8] C. Ballabriga. Vérification de contraintes temporelles strictes sur des programmes par composition d'analyses partielles. PhD thesis, 2010.

[9] C. Ballabriga and H. Cassé. Improving the first-miss computation in set-associative instruction caches. In *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2008.

[10] C. Ballabriga, H. Casse, and P. Sainrat. An improved approach for set-associative instruction cache partial analysis. In *SAC '08 : Proceedings of*

the 2008 ACM symposium on Applied computing, pages 360–367, New York, NY, USA, 2008. ACM.

[11] I. Bate and R. Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *ECRTS '04 : Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 215–222, Washington, DC, USA, 2004. IEEE Computer Society.

[12] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2) :78–101, 1966.

[13] C. Berg. PLRU cache domino effects. In F. Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Dresden*, number 06902 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), July 2006.

[14] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis : The hyperbolic bound. *IEEE Trans. Comput.*, 52(7) :933–942, 2003.

[15] J. Blieberger, T. Fahringer, and B. Scholz. Symbolic cache analysis for real-time systems. *Real-Time Syst.*, 18(2/3) :181–215, 2000.

[16] C. Burguière, J. Reineke, and S. Altmeyer. Cache-related preemption delay computation for set-associative caches—pitfalls and solutions. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

[17] C. Burguière. Modéliser la prédiction de branchement pour le calcul de temps d'exécution pire-cas. PhD thesis, 2008.

[18] L. Burkholder. The halting problem. *SIGACT News*, 18(3) :48–60, 1987.

[19] *Cache Replacement Championship, 2010.*
http ://www.jilp.org/jwac-1/.

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

[21] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[22] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for wcet analysis and layout optimizations. In *RTSS '09 : Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, pages 47–56, Washington, DC, USA, 2009. IEEE Computer Society.

[23] A. Colin and I. Puaut. Worst case execution time analysis for a processor withbranch prediction. *Real-Time Syst.*, 18(2/3) :249–274, 2000.

[24] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 37–44, Delft, The Netherlands, June 2001.

[25] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[26] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[27] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, May 2010.

[28] *Debie.*
http ://www.mrtc.mdh.se/projects/WCC08/doku.php ?id=bench :debie1.

[29] J. Deverge and I. Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Proc. of the 19th Euromicro Conference on Real-Time Systems*, pages 179–190, Pisa, Italy, July 2007.

[30] J.-F. Deverge. Contributions à l'analyse du comportement temporel de la hiérarchie mémoire pour l'estimation du pire temps d'exécution. PhD thesis, 2008.

[31] H. Dybdahl and P. Stenström. Enhancing last-level cache performance by block bypassing and early miss determination. In *Asia-Pacific Computer Systems Architecture Conference*, pages 52–66, 2006.

[32] J. Engblom. Processor pipelines and static worst-case execution time analysis. PhD thesis.

[33] M. Farrens, G. Tyson, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, 1995.

[34] C. Ferdinand. Cache behavior prediction for real-time systems. PhD thesis, 1997.

[35] C. Ferdinand and 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*, pages 16–30. Springer-Verlag, 1998.

[36] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations : a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4) :703–746, 1999.

[37] J. Goossens, S. Baruah, and S. Funk. Real-time scheduling on uniform multiprocessors. In *Proceedings of the 10th International Conference on Real-Time Systems, Paris, France*, pages 189–204, 2002.

[38] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, Washington D.C., USA, Dec. 2009.

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

[40] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th Real-Time Systems Symposium*, pages 456–466, Barcelona, Spain, Dec. 2008.

[41] D. Hardy and I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *Proceedings of the 17th Real-Time and Network Systems*, pages 45–54, Paris, France, Oct. 2009.

[42] D. Hardy and I. Puaut. Wcet analysis of instruction cache hierarchies. *Journal of Systems Architecture*, 2010, To appear.

[43] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE, vol.9, no7*, 2003.

[44] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12) :1612–1630, 1989.

[45] M. J. Irwin. Shared caches in multicores : The good, the bad, and the ugly. In *ISCA'10 : Proceedings of the 37th annual International Symposium on Computer and Architecture*, page 234, 2010.

[46] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a) :364–373, 1990.

[47] J. L .R . Ford and D. R. Fulkerson. Maximal flow through a network, can. j. math. 8 (1956), 399-404.

[48] M. Lee, S. L. Min, H. Shin, C. S. Kim, and C. Y. Park. Threaded prefetching : A new instruction memory hierarchy for real-timesystems. *Real-Time Syst.*, 13(1) :47–65, 1997.

[49] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In N. Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[50] B. Lesage, D. Hardy, and I. Puaut. Shared data cache conflicts reduction for WCET computations in multi-core architectures. In *Proceedings of the 18th Real-Time and Network Systems*, Toulouse, France, Nov. 2010, To appear.

[51] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET estimation. *Real-Time Systems Journal*, 34(3), Nov. 2006.

[52] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS '09 : Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, pages 57–67, Washington, DC, USA, 2009. IEEE Computer Society.

[53] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In R. Gerber and T. Marlowe, editors, *LCTES '95 : Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, volume 30, pages 88–98, New York, NY, USA, Nov. 1995.

[54] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software : beyond direct mapped instruction caches. In *RTSS '96 : Proceedings of the 17th IEEE Real-Time Systems Symposium*, page 254, Washington, DC, USA, 1996. IEEE Computer Society.

[55] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, 1973.

[56] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven cache-based procedure positioning optimizations. In *ECRTS '08 : Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 321–330, Washington, DC, USA, 2008. IEEE Computer Society.

[57] P. Lokuciejewski, H. Falk, P. Marwedel, and H. Theiling. WCET-driven, code-size critical procedure cloning. In *SCOPES '08 : Proceedings of the 11th international workshop on Software and compilers for embedded systems*, pages 21–30, New York, NY, USA, 2008. ACM.

[58] P. Lokuciejewski and P. Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for WCET minimization. In *ECRTS '09 : Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, pages 35–44, Washington, DC, USA, 2009. IEEE Computer Society.

[59] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Syst.*, 17(2-3) :183–207, 1999.

[60] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 255–262, 1999.

[61] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, pages 12–21, 1999.

[62] *WCET benchmarks.*
http ://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[63] A. Marref and G. Bernat. Towards predicated WCET analysis. In R. Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. also published in print by Austrian Computer Society (OCG) under ISBN 978-3-85403-237-3.

[64] S. McFarling. Program optimization for instruction caches. *SIGARCH Comput. Archit. News*, 17(2) :183–191, 1989.

[65] P. Michaud. Replacement policies for shared caches on symmetric multicores : a programmer-centric point of view. Research Report RR-6734, INRIA, 2008.

[66] *MIPS Application Binary Interface.*
http ://www.sco.com/developers/devspecs/mipsabi.pdf.

[67] F. Mueller. Static cache simulation and its applications. PhD thesis, 1994.

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

[69] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems Journal*, 18(2-3) :217–247, 2000.

[70] *Nachos.*
http ://www.cs.washington.edu/homes/tom/nachos/.

[71] H. S. Negi, T. Mitra, and 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*, pages 201–206, New York, NY, USA, 2003. ACM.

[72] H. Ozaktas, K. Heydemann, C. Rochange, and H. Cassé. Impact of Code Compression on Estimated Worst-Case Execution Times. In *International Conference on Real-Time and Network Systems (RTNS), Paris*, pages 55–64, october 2009.

[73] M. Paolieri, E. Qui nones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In

ISCA '09 : Proceedings of the 36th annual international symposium on Computer architecture, pages 57–68, New York, NY, USA, 2009. ACM.

[74] T. Piquet. Gestion consciente du contenu de la hiérarchie mémoire. PhD thesis, 2008.

[75] T. Piquet, O. Rochecouste, and A. Seznec. Exploiting single-usage for effective memory management. In *ACSAC '07 : Proceedings of the 12th Asia-Pacific conference on Advances in Computer Systems Architecture*, pages 90–101. Springer-Verlag, 2007.

[76] G. N. Prasanna. Cache structure and method for improving worst case execution time. Patent Pending, 2001. US 6,272,599 B1.

[77] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, Dresden, Germany, July 2006.

[78] I. Puaut and 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*, page 114. IEEE Computer Society, 2002.

[79] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real Time Systems Journal*, 18(2-3) :115–128, May 2000. Guest Editorial.

[80] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems Journal*, 1(2) :159–176, 1989.

[81] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph based approach. In *Proceedings of IEEE Real-Time Systems Symposium*, volume 13, pages 67–91, 1997.

[82] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA '07 : Proceedings of the 34th annual international symposium on Computer architecture*, pages 381–391, New York, NY, USA, 2007. ACM.

[83] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *RTAS '05 : Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 148–157, Washington, DC, USA, 2005. IEEE Computer Society.

[84] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *RTAS '06 : Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, Washington, DC, USA, 2006. IEEE Computer Society.

[85] J. Reineke and D. Grund. Relative competitive analysis of cache replacement policies. In *LCTES '08 : Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 51–60, New York, NY, USA, 2008. ACM.

[86] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2) :99–122, 2007.

[87] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[88] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953.

[89] C. Rochange and P. Sainrat. Code padding to improve the WCET calculability. In *International Conference on Real-Time and Network Systems (RTNS), Poitiers*, pages 159–168, may 2006.

[90] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS '07 : Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[91] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *LCTES '09 : Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 80–89, New York, NY, USA, 2009. ACM.

[92] J. E. Sasinowski and J. K. Strosnider. A dynamic programming algorithm for cache memory partitioning for real-time systems. *IEEE Trans. Comput.*, 42(8) :997–1001, 1993.

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

[94] R. Sen and 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*, pages 203–212. ACM, 2007.

[95] A. Seznec. A case for two-way skewed-associative caches. In *ISCA '93 : Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178, New York, NY, USA, 1993. ACM.

[96] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3) :473–530, 1982.

[97] J. Staschulat. Instruction and data cache timing analysis in fixed-priority preemptive real-time systems. PhD thesis, 2006.

[98] J. Staschulat, S. Schliecker, and 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*, pages 41–48, Washington, DC, USA, 2005. IEEE Computer Society.

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

[100] C. P. Thacker. Improving the future by examining the past. In *ISCA'10 : Proceedings of the 37th annual International Symposium on Computer and Architecture*, page 348, 2010.

[101] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems Journal*, 18(2-3) :157–179, 2000.

[102] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *SIGMETRICS '03 : Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282. ACM, 2003.

[103] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Syst.*, 21(3) :241–268, 2001.

[104] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *QSIC '05 : Proceedings of the Fifth International Conference on Quality Software*, pages 295–306, Washington, DC, USA, 2005. IEEE Computer Society.

[105] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Syst.*, 17(2-3) :209–233, 1999.

[106] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

[107] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7) :966–978, 2009.

[108] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, pages 281–292, 2005.

[109] J. Yan and W. Zhang. Wcet analysis of instruction caches with prefetching. *SIGPLAN Not.*, 42(7) :175–184, 2007.

[110] J. Yan and 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*, pages 80–89, 2008.

[111] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving WCET by applying worst-case path optimizations. *Real-Time Syst.*, 34(2) :129–152, 2006.

[112] Y. Zheng, B. T. Davis, and M. Jordan. Performance evaluation of exclusive cache hierarchies. In *ISPASS '04 : Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–96, Washington, DC, USA, 2004. IEEE Computer Society.

Table des figures

Résumé

Les systèmes temps-réel strict sont soumis à des contraintes temporelles dont le non respect peut entraîner des conséquences économiques, écologiques, humaines catastrophiques. Le processus de validation, garantissant la sûreté de ces logiciels en assurant le respect de ces contraintes dans toutes les situations possibles y compris le pire cas, se base sur la connaissance à priori du pire temps d'exécution de chacune des tâches du logiciel. Cependant, l'obtention de ce pire temps d'exécution est un problème difficile pour les architectures actuelles, en raison des mécanismes matériels complexes pouvant amener une variabilité importante du temps d'exécution.

Ce document se concentre sur l'analyse du comportement temporel pire cas des hiérarchies de mémoires cache, afin de déterminer leur contribution au pire temps d'exécution. Plusieurs approches sont proposées afin de prédire et d'améliorer le pire temps d'exécution des tâches s'exécutant sur des processeurs multi-cœurs disposant d'une hiérarchie de mémoires cache avec des niveaux partagés entre les différents cœurs de calculs.

Abstract

Hard real-time systems are subject to timing constraints and failure to respect them can cause economic, ecological or human disasters. The validation process which guarantees the safety of such software, by ensuring the respect of these constraints in all situations including the worst case, is based on the knowledge of the worst case execution time of each task. However, determining the worst case execution time is a difficult problem for modern architectures because of complex hardware mechanisms that could cause significant execution time variability.

This document focuses on the analysis of the worst case timing behavior of cache hierarchies, to determine their contribution to the worst case execution time. Several approaches are proposed to predict and improve the worst case execution time of tasks running on multicore processors with a cache hierarchy in which some cache levels are shared between cores.