THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

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

Présentée et soutenue par *Clément BALLABRIGA*
Le *3 septembre 2010*

Titre :
Vérification de contraintes temporelles strictes
sur des programmes par composition d'analyses partielles

JURY
Isabelle PUAUT (rapporteure)
Philippe CLAUSS (rapporteur)
Pascal SAINRAT (directeur de thèse)
Louis FERAUD (examinateur)
Hugues CASSÉ (examinateur)
Matthieu MARTEL (examinateur)

Ecole doctorale : *ED MITT*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse*
Directeur(s) de Thèse : *Pascal SAINRAT*
Rapporteurs : *Isabelle PUAUT, Philippe CLAUSS*

Table des matières

Résumé

Les systèmes temps-réel critiques, de plus en plus utilisés dans l'industrie, doivent obéir à des contraintes temporelles strictes, sans quoi les conséquences peuvent être désastreuses. Le respect de ces contraintes temporelles doit être prouvé, et le calcul de pire temps d'exécution (WCET ou Worst Case Execution Time) joue un rôle important dans le processus de validation.

Le calcul de WCET par analyse statique est traditionnellement réalisé sur un programme entier. Cette approche présente deux inconvénients principaux. Premièrement la plupart de ces méthodes de calcul voient leur temps d'exécution augmenter de manière non linéaire par rapport à la taille de la tâche à analyser, deuxièmement cette approche est problématique lorsque la tâche à analyser est constituée de plusieurs composants, dont certains ne sont pas disponibles au moment de l'analyse.

Pour ces raisons, il est nécessaire de développer une méthode d'analyse partielle pour pouvoir analyser chaque partie d'un programme séparément (produisant un résultat de WCET partiel associé à chaque partie), et pour pouvoir ensuite combiner tous ces résultats partiels pour obtenir le WCET de la tâche. Nous nous intéressons principalement au calcul de WCET par analyse statique avec la méthode IPET. Cette méthode fait intervenir plusieurs analyses pour modéliser le flot de contrôle du programme ainsi que les effets temporels liés au matériel utilisé, le résultat de ces analyses est un système de contraintes ILP qui est ensuite résolu pour obtenir le WCET. Ces analyses doivent toutes être adaptées pour faire de l'analyse partielle pour le calcul de WCET complet, et le problème de la résolution ILP doit aussi être traité.

Nous commençons par traiter le problème de l'analyse du cache d'instructions, nous nous basons sur une analyse de cache existante pour l'améliorer et fournir une version apte à faire de l'analyse partielle. Nous adaptons également une méthode d'analyse du prédicteur de branchements pour la rendre compatible avec IPET, et nous proposons une version utilisable pour l'analyse partielle. Nous présentons ensuite une analyse partielle pour l'estimation des bornes de boucles, ainsi qu'une méthode pour segmenter le système ILP à résoudre en plusieurs parties sans affecter le WCET obtenu. Enfin, grâce à ces analyses que nous avons adaptées, nous proposons une méthode complète d'analyse partielle (et de composition) pour le calcul de WCET, tout en fournissant un cadre général facilitant la prise en compte de nouvelles analyses. Des résultats expérimentaux sont fournis pour valider notre approche, montrant un gain de temps important et un pessimisme peu élevé du WCET.

Chapitre 1

Introduction

De plus en plus, en avionique comme en automobile, des systèmes temps-réel sont utilisés pour des aspects critiques (commandes de vol, gestion des freins, anti-dérapage, régulateur de vitesse, etc). Ces systèmes doivent obéir à des contraintes temporelles strictes, sans quoi les conséquences peuvent être désastreuses. C'est pourquoi il est nécessaire de prouver que ces contraintes temporelles seront bien respectées. Le calcul de pire temps d'exécution (WCET ou Worst Case Execution Time) joue un rôle important lors de la preuve du respect de ces contraintes temporelles : il permet d'estimer le temps maximal d'exécution d'une tâche du système temps-réel, en tenant compte des informations de flot de la tâche, et des effets liés au matériel (processeur, mémoire, cache, etc.) qui exécutera cette tâche.

Nous nous intéresserons principalement au calcul de WCET par analyse statique. Le calcul de WCET par analyse statique est traditionnellement réalisé sur un programme entier. Cette approche présente deux inconvénients principaux. Premièrement la plupart de ces méthodes de calcul voient leur temps d'exécution augmenter de manière non linéaire par rapport à la taille de la tâche à analyser, deuxièmement cette approche est problématique lorsque la tâche à analyser est constituée de plusieurs composants, dont certains ne sont pas disponibles au moment de l'analyse.

Pour ces raisons, il est nécessaire de développer une méthode pour pouvoir analyser chaque partie d'un programme séparément (produisant un résultat de WCET partiel associé à chaque partie), et pour pouvoir ensuite combiner tous ces résultats partiels pour obtenir le WCET de la tâche.

1.1 Contexte

Dans cette partie, nous présentons le contexte dans lequel se situe les travaux exposés dans le présent document. Nous introduisons en particulier les systèmes temps-réel, et le concept de pire temps d'exécution (WCET ou Worst Case Execution Time).

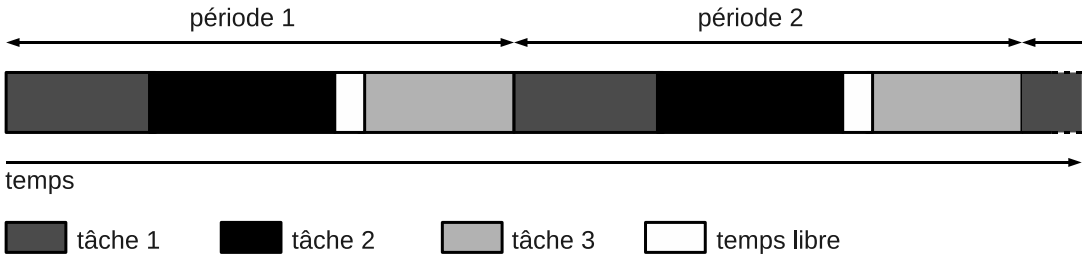


FIGURE 1.1: Les tâches d'un système temps-réel

1.1.1 Les systèmes temps-réel stricts

La conception de systèmes temps-réel stricts fait appel à l'analyse d'ordonnancement, cette analyse permet de planifier l'exécution des tâches de manière à respecter les contraintes temporelles. L'analyse d'ordonnancement nécessite de connaître le WCET de chaque tâche du système. Dans cette partie, nous présentons rapidement les systèmes temps-réel stricts.

Un système temps-réel est composé d'un ensemble de tâches, lancées - périodiquement ou bien en réponse à certains évènements - par un ordonnanceur. Dans un système temps-réel strict, ces tâches sont soumises à des contraintes temporelles strictes, c'est-à-dire qu'elles ont des échéances qu'elles doivent absolument respecter. Ce type de système est utilisé dans les cas où le non-respect des échéances peut avoir des conséquences catastrophiques.

Par exemple, le système représenté sur la figure 1.1 est composé de tâches lancées périodiquement : il est évident que chaque tâche doit être terminée lorsque la tâche suivante doit être lancée car il n'y a pas de préemption (une tâche non terminée ne peut pas être interrompue par une autre). Dans cet exemple simple, chaque tâche a la même période, mais ceci peut être différent dans la réalité. Nous nous intéressons principalement à des systèmes du même type que celui représenté dans la figure 1.1 .

Pour concevoir un tel système, une fois connue la liste des tâches ainsi que leur période et échéance, il faut trouver un ordonnancement qui est compatible avec les contraintes temporelles de chacune d'elles.

1.1.2 Le pire temps d'exécution (WCET)

Comme montré dans la section précédente, l'analyse d'ordonnancement se base sur le temps d'exécution de chaque tâche : c'est pourquoi il est nécessaire de connaître une estimation du WCET de chaque tâche au préalable. Le WCET de la tâche dépend du programme, et du matériel utilisé pour exécuter celui-ci (l'architecture cible). Sous-estimer le WCET (être « optimiste ») permettrait de mettre en place un ordonnancement qui pourrait violer les contraintes temporelles. Sur-estimer le WCET (être « pessimiste ») provoquerait le sur-dimensionnement du matériel et le gaspillage de ressources (figure 1.2).

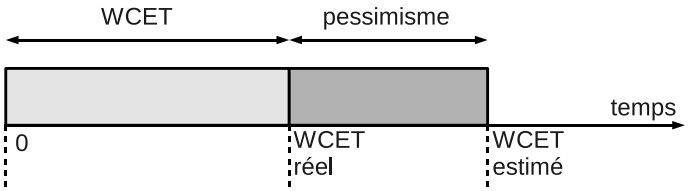


FIGURE 1.2: Le WCET d'une tâche

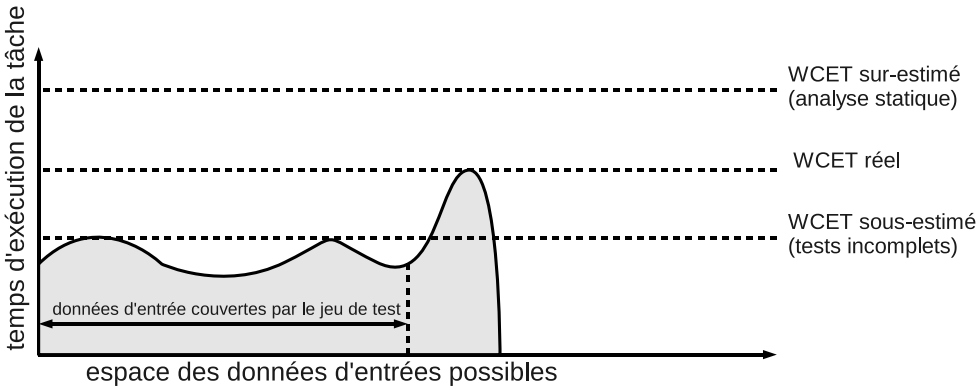


FIGURE 1.3: Comparaison des approches de calcul de WCET

Il existe deux grandes catégories de méthodes d'estimation du WCET : (1) les méthodes dynamiques, et (2) les méthodes statiques.

Les méthodes dynamiques d'estimation du WCET se basent sur les tests, ces types de méthodes vont généralement essayer d'exécuter le programme dans le plus de cas possibles pour en mesurer son pire temps d'exécution. Ce genre d'approche a pour inconvénient le fait qu'il est difficile (sauf pour des programmes très simples) d'être sûr de prendre en compte tous les cas possibles, ce qui peut produire une sous-estimation du WCET [16].

Dans les méthodes de calcul de WCET par analyse statique, l'outil de calcul de WCET analyse le programme (source et/ou binaire, ainsi que l'architecture cible) et produit un WCET, sans avoir à exécuter réellement le programme analysé. La principale différence avec les méthodes dynamiques, est qu'il n'y a pas de test. Donc, avec ce type de méthodes, on ne peut, en principe, pas avoir de sous-estimation du WCET, mais les analyses peuvent être complexes et/ou introduire du pessimisme (figure 1.3).

1.2 Motivations, et problème de l'approche actuelle

Dans cette partie, nous expliquons ce qui a motivé nos travaux. Nous indiquons quels sont les problèmes des approches existantes, et comment nous prévoyons de les résoudre.

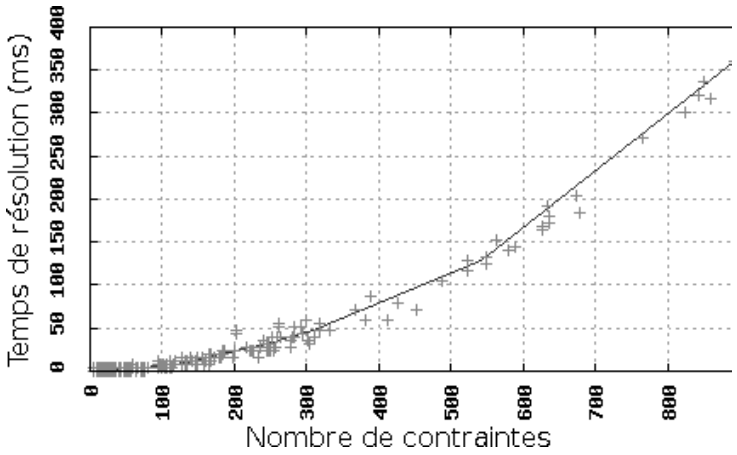


FIGURE 1.4: Temps de calcul en fonction du nombre de contraintes

1.2.1 Limitations de l'approche actuelle de calcul du WCET

Nous nous intéressons uniquement à l'estimation du WCET par analyse statique, car elle produit obligatoirement des résultats sûrs (c'est-à-dire que le WCET peut être sur-estimé mais non sous-estimé), ce qui est exigé dans le cadre des systèmes temps réel stricts. Traditionnellement, le calcul du WCET est réalisé sur un programme complet. Cette approche présente quelques inconvénients, qui sont présentés ci-dessous.

L'estimation du WCET d'un programme fait intervenir un nombre assez conséquent d'analyses (il y a les analyses liées au code de la tâche, et les analyses liées aux différents éléments d'architecture à prendre en compte tels que le cache, le pipeline du processeur, etc.), la plupart de ces analyses sont assez coûteuses en temps d'exécution et d'utilisation de mémoire, et ce coût a souvent une croissance non linéaire par rapport à la taille du programme à analyser.

Les programmes temps-réel utilisés dans l'industrie deviennent de plus en plus gros, donc le temps de calcul risque de devenir un problème important. Des mesures ont été faites avec une analyse relativement réaliste : la figure 1.4 montre le temps de calcul du WCET en fonction du nombre de contraintes utilisées dans la méthode IPET (voir section). Le nombre de contraintes croît avec la taille du programme à analyser.

De plus en plus de programmes temps-réel utilisent des COTS (Components Off The Shelf). Ce sont des composants pouvant être réutilisés d'un projet à l'autre, et souvent créés et distribués indépendamment du programme principal (figure 1.5). De ce fait, lorsqu'on veut calculer le WCET du programme principal, si ce programme utilise des COTS, ceux-ci devront être analysés en même temps que le programme principal puisque son temps d'exécution est influencé par le temps passé dans les COTS.

Ceci veut dire qu'on doit re-analyser les COTS à chaque fois qu'ils sont utilisés dans un programme, ce qui fait perdre du temps (ceci rejoint le problème exposé précédemment). De plus, si les COTS sont propriétaires et développés par une entité différente de celle qui développe le programme principal, le développeur du programme principal pourra

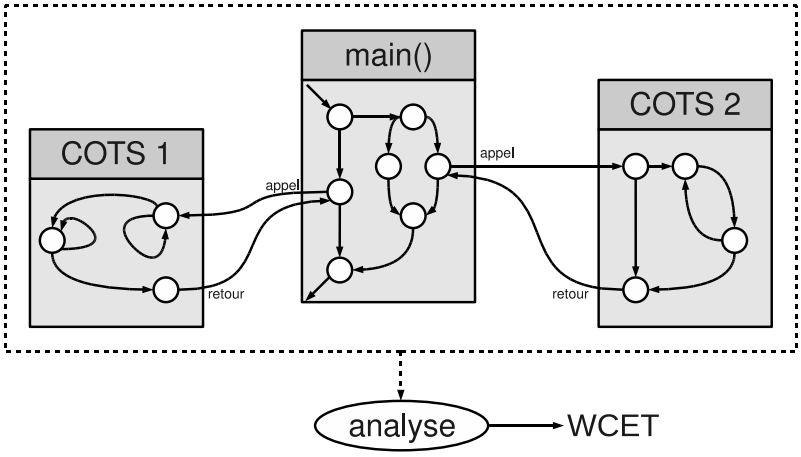


FIGURE 1.5: Calcul de WCET par analyse monolithique

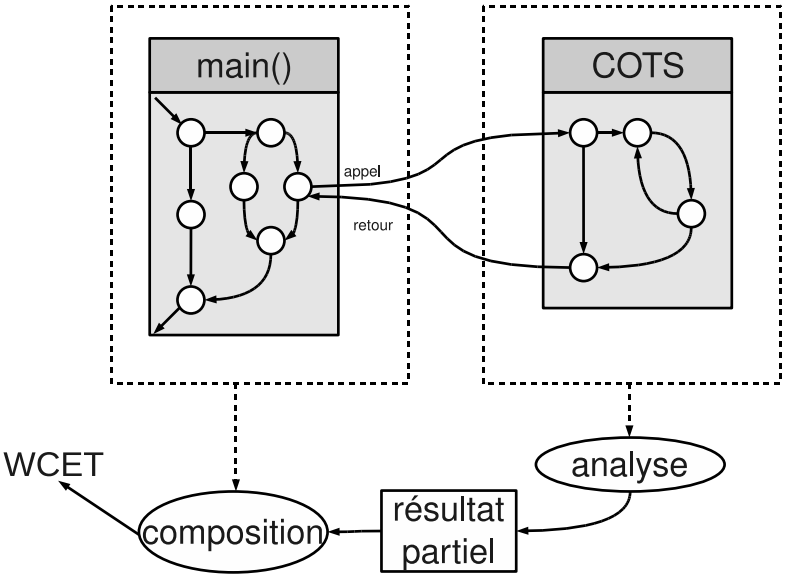


FIGURE 1.6: Calcul de WCET par analyse partielle

avoir du mal à calculer son WCET puisqu'il lui manquera des informations sur les COTS (certaines analyses de WCET nécessitent le code source).

1.2.2 Une solution : l'analyse partielle

Pour tenir compte de ces limitations, il est nécessaire de développer une méthode d'analyse partielle de composants logiciels. Cette analyse partielle a pour but de produire un résultat partiel, qui peut ensuite être instancié lors de l'analyse du programme principal utilisant le composant, permettant d'aboutir à un résultat complet.

Pour le problème du temps de calcul, diviser le programme en composants permet de gagner du temps pour deux raisons : premièrement, parce que le temps d'analyse croît de manière non linéaire par rapport à la taille du code analysé, et deuxièmement parce

que les composants peuvent être utilisés en plusieurs endroits, ce qui permet de faire de la réutilisation de résultats partiels.

Ceci répond aussi au problème des COTS propriétaires : le développeur du COTS peut le livrer accompagné de son résultat partiel, tel que présenté dans la figure 1.6. Ensuite, le développeur du programme principal peut instancier ce résultat partiel dans son analyse pour prendre en compte la contribution au WCET du temps passé dans le COTS.

Il faut donc (1) trouver comment découper un programme en composants (sauf dans le cas des COTS, qui constituent en quelque sorte un découpage pré-existant imposé par la structure du projet), (2) spécifier ce que constitue un résultat partiel, et comment le produire, et (3) savoir instancier le résultat partiel au sein de l'analyse du programme principal.

1.3 Liste des principales contributions

Nos travaux sont principalement axés sur l'analyse partielle, dans le cadre du calcul de WCET. Afin de réaliser ces travaux, nous avons dû étudier plusieurs facettes du calcul du WCET. Nous nous sommes intéressés en premier lieu à la prise en compte du cache d'instructions lors du calcul du WCET.

C'est pourquoi notre premier objectif a été l'adaptation et l'amélioration de la technique d'analyse du cache d'instructions par interprétation abstraite de C. Ferdinand et al., afin de rendre cette analyse plus rapide et plus précise, et de la rendre plus adaptée à l'analyse partielle. Ceci est exposé dans le chapitre 3, où il est également présenté des solutions à d'autres problèmes proches du problème du cache, comme le *row buffer* (le *row buffer* peut être considéré comme une sorte de cache situé entre le cache d'instruction et la RAM dynamique, dans le cas que nous avons étudié).

Ensuite, nous avons développé une méthode d'analyse partielle du cache d'instructions qui essaie de corriger les défauts présents dans d'autres analyses du même type, et ce à partir de l'analyse développée dans le chapitre 3. Cette méthode a été testée et comparée à d'autres approches dans le même domaine. Ces travaux sont présentés dans le chapitre 4.

Nous avons aussi adapté une méthode d'analyse des bornes de boucles à l'analyse partielle, ainsi qu'une méthode d'analyse du prédicteur de branchements. Ces analyses partielles ont été réalisées à partir des analyses (non partielles) existantes, respectivement, de M. de Michiel et al. [24], ainsi que de I. Puaut. et al. [15] exposées dans le chapitre 2. Ces analyses partielles sont exposées dans le chapitre 5.

Nous avons aussi développé une méthode d'accélération de la méthode IPET (phase finale, et la plupart du temps coûteuse, employée dans de nombreuses approches du calcul de WCET par analyse statique) basée sur la subdivision du programme à analyser en régions indépendantes. Ceci est présenté dans le chapitre 6.

Enfin, nous avons repris et généralisé les approches mentionnées ci-dessus, afin de proposer une méthode complète d'analyse partielle du calcul de WCET, fonctionnant avec les analyses partielles déjà exposées, et prévue pour être extensible à des analyses

partielles futures. Ceci est présenté dans le chapitre 7.

Tout ceci a été implémenté avec OTAWA un outil de calcul de WCET développé à l'IRIT, qui a donc été enrichi par l'ajout de ces nouvelles méthodes d'analyse. Une discussion plus approfondie sur OTAWA et sur les contributions réalisées pour cet outil sera faite dans le chapitre 8.

1.4 Plan et organisation de la thèse

Nous commençons par présenter l'état de l'art de l'analyse de WCET par analyse statique, dans le chapitre 2. Nous poursuivons par l'exposé de nos travaux sur la hiérarchie mémoire, concernant principalement le cache d'instructions, dans le chapitre 3. Dans le chapitre 4, nous exposons notre méthode pour l'analyse partielle du cache d'instructions, puis dans le chapitre 5, celle l'analyse partielle d'autre effets liés au WCET. Nous présentons notre approche sur la subdivision du programme à analyser en régions indépendantes dans le chapitre 6. La généralisation de notre approche d'analyse partielle est présentée dans la section 7. Le chapitre 8 présente les aspects techniques de l'implémentation des méthodes qui ont été développées. Enfin, nous concluons dans le chapitre 9.

Chapitre 2

WCET par analyse statique

Cette partie est une introduction au calcul du WCET par analyse statique. Nous présentons d'abord certaines techniques générales de représentation et d'analyse de programme, puis nous étudions plus particulièrement le problème du WCET.

2.1 Analyse statique

Dans cette section, nous donnons les pré-requis utiles pour comprendre les techniques de calcul du WCET par analyse statique.

2.1.1 Représentations des programmes

Un grand nombre d'analyses dans l'estimation du WCET travaillent sur une représentation du programme binaire sous forme de Graphe de Flot de Contrôle (CFG). Un CFG est un graphe orienté (N, E) où l'ensemble des nœuds N est l'ensemble des *blocs de base* (BB) du programme. Un bloc de base représente une séquence d'instructions ayant un seul point d'entrée, et un seul point de sortie (en d'autre termes, il ne peut pas y avoir d'instructions de branchement sauf peut-être à la fin, et il ne peut pas y avoir d'étiquette sauf peut-être au début). E représente l'ensemble des arcs, un arc (bb_1, bb_2) appartient à E si le flot de contrôle peut passer du bloc de base bb_1 au bloc de base bb_2. Ceci peut arriver soit lorsque deux BB sont en séquence, soit lorsque il existe une instruction de branchement entre les deux (exemple sur la figure 2.1). Un *chemin* de bb_1 à bb_n dans le CFG est une séquence de blocs de base, notée $(bb_1, bb_2, bb_3, ..., bb_{n-1}, bb_n)$, telle que $(bb_1, bb_2) \in E$, $(bb_2, bb_3) \in E$, ..., $(bb_{n-1}, bb_n) \in E$.

Un bloc de base spécial *entrée*, n'ayant pas de prédécesseurs, représente le point d'entrée du programme, et un un autre bloc de base spécial *sortie*, n'ayant pas de successeurs, représente la sortie. Le bloc *sortie* est un passage obligé pour sortir du programme : si un programme a plusieurs points de sortie aux blocs de base bb_1 et bb_2 (par exemple plusieurs *return* dans une fonction), alors on crée un bloc *sortie* virtuel ne comportant aucune instruction, et on crée des arcs $(bb_1, sortie)$ et $(bb_2, sortie)$. Un algorithme de construction de CFG à partir d'un programme donné peut être trouvé dans [1].

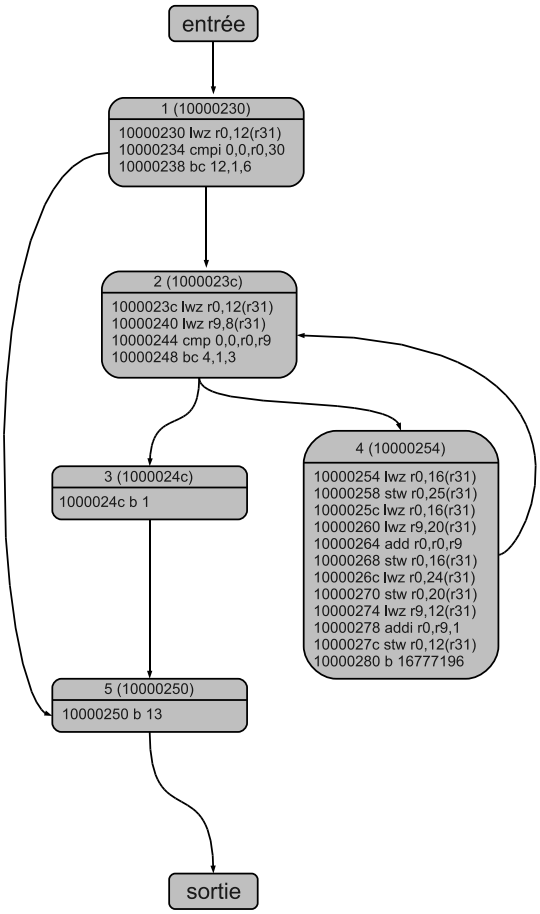


FIGURE 2.1: Un exemple de graphe de flot de contrôle

2.1.2 La domination

Les boucles du programme sont visibles dans le CFG. Toutefois, pour pouvoir exprimer la définition d'une boucle dans le CFG, nous devons d'abord définir la notion de blocs de base *dominants*. Un bloc de base bb_1 domine le bloc de base bb_2 (ce qu'on notera $bb_1\, dom\, bb_2$) si tout chemin $(entrée, ..., bb_2)$ passe par bb_1. Le bloc de base bb_1 post-domine bb_2 (ce qu'on notera $bb_1\, postdom\, bb_2$) si tout chemin $(bb_2, ..., sortie)$ passe par bb_1. Par exemple, dans la figure 2.2, le bloc de base 4 domine les blocs 5, 6 et 7 (en plus de lui-même). Par contre, le bloc de base 2 ne domine aucun bloc (à part lui-même), car il existe un chemin passant par le bloc 3 (et donc ne passant pas par le bloc 2) pour atteindre les blocs 5, 6, et 7.

La domination (ainsi que la post-domination) est une relation d'ordre partiel large : $\forall bb$ on a $bb\, dom\, bb$ (réflexivité), $\forall bb_1, bb_2 \in N$ on a $(bb_1\, dom\, bb_2) \wedge (bb_2\, dom\, bb_1) \implies bb_1 = bb_2$ (antisymétrie), $\forall bb_1, bb_2, bb_3$ on a $(bb_1\, dom\, bb_2) \wedge (bb_2\, dom\, bb_3) \implies (bb_1\, dom\, bb_3)$, et de plus on peut avoir deux blocs bb_1 et bb_2 de telle sorte que $\neg(bb_1\, dom\, bb_2)$ et $\neg(bb_2\, dom\, bb_1)$. Le bloc de base d'entrée domine tous les blocs de base du CFG, et le bloc de base de sortie post-domine également tous les blocs de base. Avant de pouvoir identifier les boucles du CFG, il faut calculer la liste des dominants pour chaque bloc de base. Un algorithme pour

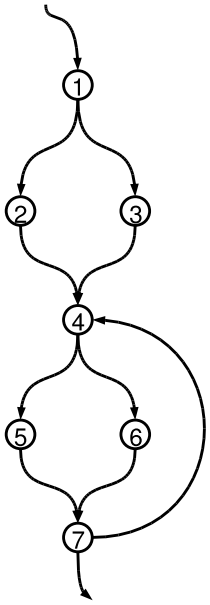


FIGURE 2.2: Domination

réaliser ceci de manière efficace peut être trouvé dans [17].

2.1.3 Les boucles dans le CFG

Une boucle existe lorsqu'on a un ou plusieurs arcs de type (bb_i, H_j) tels que $H_j \, dom \, bb_i$. Dans ce cas, H_j est appelé la tête de boucle, et les arcs (bb_i, H_j) sont les *arcs retour* de cette boucle. On désignera par L_j, également, la boucle ayant pour tête le bloc de base H_j. Les autres arcs (bb_i, H_j) tels que $\neg(H_j \, dom \, bb_i)$ sont les *arcs d'entrée* de la boucle. En d'autres mots, les arcs d'entrée correspondent aux entrées dans la boucle depuis l'extérieur de la boucle, tandis que les arcs retour correspondent au passage vers l'itération suivante de la boucle. Par ailleurs, la tête de boucle domine tous les blocs de base de la boucle. Un bloc de base bb_i est dans la boucle L_j s'il existe un chemin (bb_i, ..., bb_j, H_j) tel que $bb_j \neq H_j$ et $(bb_j, H_j) \in E$ est un arc retour. Un arc de sortie de L_j est un arc qui va d'un bloc de base dans L_j vers un bloc de base qui n'est pas dans L_j.

Il faut noter que cette définition des boucles ne prend en compte que les boucles n'ayant pas plusieurs entrées, autrement dit les boucles irréductibles. Une méthode pour tester rapidement si un CFG est réductible est donné par R. Tarjan dans [84].

Dans la suite de ce document, nous supposerons qu'il n'y a pas de boucles irréductibles. En effet, on peut toujours transformer les boucles irréductibles en boucles réductibles (par duplication de blocs de base), une telle transformation est d'ailleurs utilisé dans OTAWA [11]. Différentes approches pour le traitement des boucles irréductibles ont été proposées dans [83, 75, 42]. Des approches pour détecter, en plus des boucles, d'autre types de structures (conditions, etc.) ont également été proposées [12].

2.1.4 L'analyse de flot de données itérative (DFA)

L'analyse de flot de données itérative (DFA - Data Flow Analysis) est une méthode permettant de calculer un ensemble d'états possibles pour un ensemble de points d'un programme (typiquement, avant et après chaque bloc de base). Ce type d'analyse est utilisé la plupart du temps dans le cadre de l'optimisation de code pour les compilateurs, mais peut aussi être très utile pour l'analyse statique sur le code binaire (et donc pour le calcul de WCET). Les états sont généralement représentés par des ensembles de valeurs. Ils sont souvent optimisés sous forme d'ensemble de bits *(BitSet)* permettant de réaliser de manière performante les opérations d'union (OU bit à bit), d'intersection (ET bit à bit) et de complément (NON bit à bit) d'ensemble. Pour chaque bloc de base bb_i, deux équations DFA sont créées : $OUT_i = IN_i \cup GEN_i \cap (\neg KILL_i)$ pour exprimer l'état après le bloc de base en fonction de l'état avant, et $IN_i = \bigcup_{x \in preds(i)} OUT_x$ (où $preds(i)$ est l'ensemble des prédécesseurs de bb_i). L'ensemble GEN_i représente les bits du BitSet mis à 1 par le passage via bb_i, alors que l'ensemble $KILL_i$ représente les bits mis à 0 par le passage via bb_i. Ces équations DFA sont appliquées itérativement sur tous les blocs du CFG jusqu'à ce que l'ensemble des états à chaque point du programme converge vers un point fixe. On peut remarquer que la fonction qui permet de passer de l'état IN à l'état OUT par application du GEN et du KILL est croissante. L'état avant un bloc croît aussi en fonction de l'état après ses prédécesseurs. Donc au fur et à mesure qu'on applique itérativement les équations DFA, l'état à chaque bloc de base est croissant, c'est pourquoi l'application itérative de ces équations DFA converge vers un point fixe. Plus de détails sur la méthode du DFA itératif sont donnés dans [94, 64]. Il y est d'ailleurs également expliqué que l'ordre d'application des équations ne change pas le résultat final, mais peut influencer le temps d'exécution de l'algorithme.

Un exemple classique de DFA utilisé dans l'optimisation de code est l'analyse des définitions visibles. Dans cet exemple, le BitSet contient un bit par définition dans le programme à analyser. L'ensemble GEN d'un bloc de base provoque donc la mise à 1 des bits correspondant aux définitions qu'il produit, tandis que l'ensemble $KILL$ d'un bloc de base provoque la mise à 0 des bits correspondants aux définitions qu'il supprime. Par exemple, considérons le bloc de base comportant les définitions suivantes :

d_1: `y := 3`
d_2: `x := 4`
d_3: `y := 5`

Ce bloc de base produit (GEN) les définitions d_2 et d_3 (et pas d_1 car cette définition est masquée par d_3). Il supprime ($KILL$) toutes les définitions d_i qui ont la forme $d_i : y := ...$ ou $d_i : x := ...$. Lorsqu'un bloc de base possède deux prédécesseurs, les définitions visibles à l'entrée du bloc de base sont les définitions visibles à la sortie de chaque prédécesseur, ce qui est modélisé par l'équation $IN_i = \bigcup_{x \in preds(i)} OUT_x$. L'application répétitive des équations de DFA sur le CFG nous permettra de converger vers un point fixe, et d'obtenir les définitions visibles à chaque point du programme.

Il est à noter que, dans certaines analyses DFA, lorsqu'un bloc de base possède plusieurs prédécesseurs, on fait l'intersection des états et non l'union. Dans ce cas, l'équation $IN_i = \bigcup_{x \in preds(i)} OUT_x$ est remplacée par $IN_i = \bigcap_{x \in preds(i)} OUT_x$, mais le principe général reste le même. De même, certaines analyses DFA sont réalisées par un parcours du CFG inversé : c'est-à-dire sur un CFG dont on a au préalable inversé le sens de chaque arc.

2.1.5 Interprétation abstraite

L'interprétation abstraite est une approche introduite par P. et R. Cousot, qui a fait l'objet de plusieurs articles [21, 22, 18, 19, 20] et qui permet de prédire des propriétés sur des programmes par analyse statique. L'interprétation abstraite travaille à partir d'une abstraction des propriétés du programme, c'est-à-dire une approximation conservatrice contenant uniquement l'information utile pour l'analyse statique à réaliser. Pour une analyse donnée, on définit un treillis D (respectivement $D\prime$) représentant le domaine concret (respectivement abstrait). Chaque élément de D représente un état concret du programme. Chaque élément de D' correspond à un *ensemble* d'éléments de D, en d'autres termes un élément de $D\prime$ représente un ensemble d'états possibles durant l'exécution du programme, ce qui permet de déduire un ensemble de propriétés qui pourront être vraies lors de l'exécution. Généralement, les deux éléments spéciaux $\bot$ et $\top$ correspondent respectivement à *aucun* et à *tous* les éléments de l'ensemble D.

Une fonction monotone α de D dans $D\prime$ permet de passer du domaine concret au domaine abstrait, et une fonction monotone γ de $D\prime$ dans 2^D permet d'associer à un élément abstrait l'ensemble d'éléments concrets le représentant. Ces fonctions ne sont pas obligatoirement exactes (c'est-à-dire qu'il peut y avoir perte de précision à cause de l'abstraction), mais on doit avoir l'équivalence suivante : $\forall d \in D, d\prime \in D\prime : d\prime \geq \alpha(d) \equiv d \in \gamma(d\prime)$ (si ces inégalités sont vraies alors la relation d'abstraction δ s'applique sur d et d' $d \,\delta\, d\prime$). Soit la fonction $\alpha\prime$ de 2^D dans $D\prime$ définie tel que $\forall x \in 2^D \, \alpha\prime(x) = \bigcup_{i \in x} \alpha(i)$, les fonctions $\alpha\prime$ et γ forment une connexion de Galois [30] entre 2^D et $D\prime$. La sémantique concrète du programme est représentée par une fonction f de D dans D qui représente l'effet (d'un morceau) du programme sur le domaine D. La sémantique abstraite est représentée par une fonction f' de $D\prime$ dans $D\prime$, et on doit avoir $d \,\delta\, d\prime \implies f(d) \,\delta\, f\prime(d\prime)$. Tout ceci est illustré sur le schéma 2.3.

2.1.6 Interprétation abstraite appliquée sur le CFG

L'interprétation abstraite est assez souvent utilisées pour le calcul de WCET [86, 45, 27, 88], dans ce cadre le programme à analyser est la plupart du temps représenté sous forme de CFG. La description de l'interprétation abstraite donnée dans la section précédente est plutôt orientée vers la sémantique opérationnelle, qui est bien adaptée pour décrire le fonctionnement de l'analyse sur un CFG.

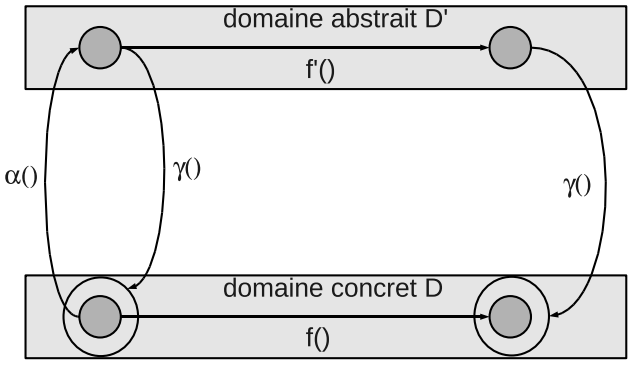


FIGURE 2.3: Abstraction et Concrétisation

On peut définir une fonction d'*update* concrète pour chaque bloc de base bb notée $u_{bb} : D \to D$. Cette fonction renvoie l'état concret après bb en fonction de l'état concret avant bb (c'est en quelque sorte la spécialisation aux CFGs de la fonction sémantique f présentée dans la section précédente). Ensuite, en employant le principe de la sémantique opérationnelle on peut définir le comportement du programme par les éléments suivants :

- Un ensemble $Z = P \times D$ d'états. L'ensemble P représente un point du programme, dans le cas du CFG ceci représentera un point situé immédiatement avant ou après un bloc de base. Appelons P_{in} l'ensemble des points du programme se situant avant un bloc de base, et P_{out} les autres. L'ensemble D représente l'état concret du programme, un élément du domaine concret.
- Un ensemble I d'états initiaux. Dans le cas du CFG, l'état initial sera $I = (p, d)$ avec p correspondant obligatoirement au point d'entrée unique du CFG, et d dépendant des conditions initiales (arguments, etc) du programme.
- Un ensemble F d'états finaux. Dans le cas du CFG, l'état final sera $F = (p, d)$ avec p correspondant forcément au point de sortie unique du CFG, et d représentant l'état concret à la sortie du programme.
- Une relation de transition $\to$ qui relie un état $z \in Z$ à son successeur $z_s \in Z$. Dans le cas du CFG, soient $z = (p, d)$, $z_s = (p_s, d_s)$; on a $z \to z_s$ si p est le point du programme avant le bloc de base bb, et que p_s est le point du programme après celui-ci, et que $d_s = u_{bb}(d)$. On a également $z \to z_s$ si p est le point du programme après le bloc de base bb_1, que p_s est le point du programme avant le bloc bb_2 et que l'arc (bb_1, bb_2) appartient au CFG.

2.1.6.1 Le cas simple : les CFGs sans boucles

Lorsqu'on fait de l'interprétation abstraite avec un CFG, on peut associer un élément du domaine abstrait $D\prime$ (*état abstrait*) à chaque point p du programme. La sémantique abstraite est implantée par une fonction monotone d'*update* abstraite $u_{bb}\prime : D\prime \to D\prime$ associée à chaque bloc de base (ceci est l'adaptation aux CFGs de la fonction $f\prime$ de sémantique abstraite donnée dans la section précédente). Pour chaque BB, la fonction $u_{bb}\prime$ donne l'état après le BB en fonction de l'état avant.

Lors de l'interprétation abstraite, on fait évoluer des états abstraits correspondant à

```
fonction(noeudEntree, etatEntree) {
 workList = {noeudEntree}
 Pour tout arc A { etat(A) := ⊥ }
 Tant que workList non vide {
  courant = recupererNoeud(workList)
  enleverNoeud(workList, courant)
  Si (courant == noeudEntree) {
   etatCourant := etatEntree
  } sinon {
   etatCourant := ⊥
   Pour tout arc d'entrée A du noeud courant {
    etatCourant := etatCourant ∪ etat(A)
   }
  }
  etatCourant := update(noeudCourant, etatCourant);
  Pour tout arc de sortie A du noeud courant {
  etat(A) := etatCourant
  ajouterNoeud(workList, destination(A))
  }
 }
}
```

FIGURE 2.4: Interprétation abstraite et CFG

des ensembles d'états. C'est pourquoi on est amené à suivre plusieurs chemins du CFG en parallèle. En effet, lorsqu'on rencontre un nœud condition avec deux arcs sortants, l'état abstrait peut être tel que les deux alternatives sont possibles. C'est pourquoi il nous faut un mécanisme pour fusionner deux états abstraits lorsqu'on rencontre une jonction entre deux chemins. Lors des jonctions entre chemins (deux BB ayant un successeur commun), la fonction utilisée est le *join* noté $j : D\prime \times D\prime \rightarrow D\prime$, cette fonction est équivalente à la plus petite borne supérieure (LUB - Least Upper Bound) commune aux deux états. Comme la fonction *join* doit être associative, il est possible de l'étendre à un nombre arbitraire d'arguments. L'état abstrait initial (correspondant au bloc de base d'entrée du CFG) est choisi de telle manière à ce qu'il soit une abstraction valide de tous les états concrets initiaux possibles. En général, si on ne sait rien des conditions initiales du programme, on choisit $\top$ comme état initial abstrait, car c'est l'état abstrait qui est, par définition, une abstraction valide pour tous les états concrets possibles. On applique successivement les fonctions $u_{bb}\prime()$ et $j\prime()$ pour propager les états depuis l'entrée du CFG vers tous les points du programme (une description détaillée de l'algorithme est donnée dans la figure 2.4).

Grâce à cette approche, on peut obtenir à chaque point du programme p, un état abstrait $d^p\prime$ tel que pour toute suite de transitions (concrètes) $i \in I \rightarrow ... \rightarrow (p, d)$ on ait $d\,\delta\,d^p\prime$. En d'autre mots, l'état abstrait $d^p\prime$ doit être une abstraction valide pour tout les états concrets pouvant arriver à ce point du programme. Si le CFG ne contient pas de boucles, il est très facile de prouver cette propriété.

En effet, par définition la propriété est vraie pour le point du programme avant le bloc de base d'entrée. Pour les autres points du programme, nous commençons par montrer

par induction que la propriété est vraie pour tout point du programme $p \in P_{in}$. Ensuite, nous nous occuperons des points $p \in P_{out}$. Pour tout autre point du programme $p \in P_{in}$ avant un bloc de base bb, on sait que $d^{p\prime} = J(d^{p1\prime}, d^{p2\prime}, ...d^{pn\prime})$, où $p1$, $p2$, ..., pn sont les points du programme à la sortie des prédécesseurs de bb. Faisons l'hypothèse que pour toute suite de transitions $i \in I \to ... \to (pi, d^{pi})$ on a $d^{pi}\,\delta\,d^{pi\prime}$ (pour i variant de 1 à n). A la fin de chacune de ces suites, on peut rajouter la transition $(pi, d^{pi}) \to (p, d^{pi})$, on obtient alors l'ensemble des suites de transitions vers le point de programme p, et pour chacune d'entre elles, on a $d^{pi}\,\delta\,d^{pi\prime}$, c'est-à-dire $\alpha(d^{pi}) \leq d^{pi\prime}$, donc on a aussi $\alpha(d^{pi}) \leq d^{p\prime}$ car $d^{p\prime}$ est l'union de tous les $d^{pi\prime}$. Donc pour l'ensemble des transitions de la forme $i \in I \to ... \to (p, d)$, on a $d\,\delta\,d^{p\prime}$.

Pour tout autre point du programme $q \in P_{out}$ se trouvant après le bloc de base bb, soit $p \in P_{in}$ le point du programme qui est juste avant bb. On a $d^{q\prime} = u_{bb\prime}(d^{p\prime})$. Par hypothèse d'induction, pour toute suite de transitions vers p, de la forme $i \in I \to ... \to (p, d)$, on a $d\,\delta\,d^{p\prime}$. Puisque le couple de fonctions $(u, u\prime)$ doit avoir la propriété $d\,\delta\,d\prime \implies u_{bb}(d)\,\delta\,u_{bb\prime}(d\prime)$ comme précisé dans la section précédente pour f et $f\prime$, alors toute suite de transitions vers q $i \in I \to ... \to (q, u(d))$ on a $u(d)\,\delta\,u\prime(d^{q\prime})$.

Ceci montre que les états abstraits associés à chaque point du programme par l'algorithme d'interprétation abstraite représentent bien un majorant (au sens de l'ordre des éléments du treillis utilisé pour le domaine abstrait, $D\prime$) des états concrets qui arrivent à ce point (dans le cas d'un CFG sans boucles).

Dans la figure 2.5, il est montré un exemple de programme sans boucle, ainsi que le CFG correspondant. Le bloc de base 1 représente l'initialisation du programme ainsi que le test de la condition. Le bloc de base 2 représente la branche *then*, tandis que le bloc de base 3 représente la branche *else.* Enfin, le bloc de base 4 représente la partie finale du programme. Grâce à ce programme nous pouvons illustrer l'interprétation abstraite sur le domaine abstrait des intervalles. Ce domaine modélise l'intervalle des valeurs possibles de chaque variable du programme. Soit V l'ensemble des variables qui existent dans le programme. Le domaine concret D est une fonction $V \to \mathbb{N}$, associant à chaque variable une valeur entière, concrète (on suppose qu'on n'utilise pas de flottants dans ce programme). Le domaine abstrait D est une fonction de type $V \to I$, où I est l'ensemble des intervalles d'entiers. Un élément de I est noté $[borne_{inf}; borne_{sup}]$ avec $borne_{inf} \in (\mathbb{N} \cup -\infty)$ et $borne_{sup} \in (\mathbb{N} \cup +\infty)$, et correspond à tout entier n tel que $borne_{inf} \leq n \leq borne_{sup}$, par conséquent la fonction de concrétisation γ est définie de telle sorte que $n \in \gamma([borne_{inf}; borne_{sup}])$ ssi $borne_{inf} \leq n \leq borne_{sup}$. La fonction d'abstraction α peut être définie de telle sorte que $\alpha(n) = [n; n]$. On voit que l'équivalence $\forall d \in D, d\prime \in D\prime : d\prime \geq \alpha(d) \equiv d \in \gamma(d\prime)$ est bien respectée, en effet si l'intervalle $d\prime = [borne_{inf}; borne_{sup}]$ inclus $\alpha(d) = [d; d]$, alors on a $borne_{inf} \leq d \leq borne_{sup}$, et donc $d \in \gamma(d\prime)$. Réciproquement, si $d \in \gamma(d\prime)$ avec $d\prime = [borne_{inf}; borne_{sup}]$, alors $borne_{inf} \leq d \leq borne_{sup}$ et donc $[borne_{inf}; borne_{sup}] \geq [d; d]$, et on a alors $d\,\delta\,d\prime$.

L'état d'entrée $\top$ correspond à la fonction $\lambda v[-\infty; +\infty]$, c'est-à-dire la fonction qui associe tout les entiers possibles à chaque variable du programme, car au début du pro-

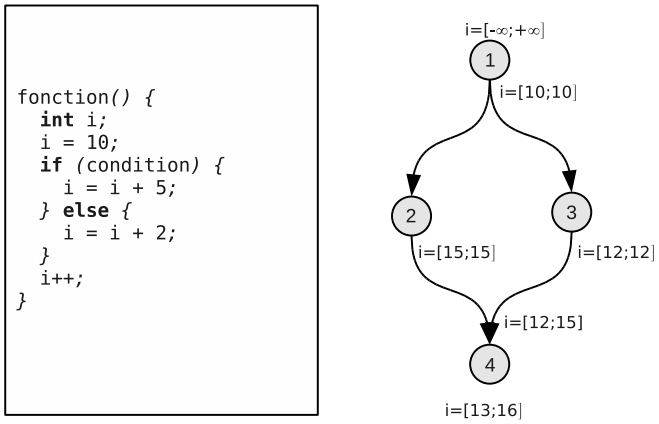


FIGURE 2.5: Interprétation abstraite et CFG

gramme on ne dispose d'aucune information sur la valeur des variables. La fonction $u\prime : D\prime \rightarrow D\prime$, associée à un bloc de base donné, modélise l'effet de ce bloc de base sur l'intervalle de chaque variable. Par exemple, le bloc de base 1 possède une fonction $u\prime$ qui prend l'état abstrait en début de programme ($\top$) et renvoie l'état abstrait $i = [10; 10]$, représentant l'initialisation de la variable i. Cet état de sortie du bloc de base 1 est utilisé comme état d'entrée des blocs de base 2 et 3. Les fonctions $u\prime$ des blocs de base 2 et 3 permettent la prise en compte, respectivement, des deux instructions $i = i + 5$ et $i = i + 2$, donnant en sortie les états abstraits $i = [15; 15]$ et $i = [12; 12]$. Le calcul de l'état abstrait avant le bloc 4 nécessite l'utilisation de la fonction $j\prime$ car il y a une jonction de chemins. Rappelons que la fonction $j\prime$ doit conserver la validité de l'abstraction. Une fonction $j\prime$ respectant cette abstraction pourrait être définie de telle sorte que $j\prime([borne^1_{inf}; borne^1_{sup}][borne^2_{inf}; borne^2_{sup}]) = [min(borne^1_{inf}; borne^2_{inf}); max(borne^1_{sup}; borne^2_{sup}])$. Utilisons cette fonction pour réaliser la jonction des chemins devant le bloc de base 4, le résultat est $j\prime(i = [15; 15], i = [12; 12]) = (i = [12; 15])$. On constate que ce résultat final représente bien l'ensemble des possibilités d'état concrets pour la variable i.

2.1.6.2 Le cas général : les CFGs pouvant comporter des boucles

Dans le cas où le CFG comporte des boucles, nous supposerons que ces boucles ne comportent qu'une entrée, c'est-à-dire qu'il n'y a pas de boucle irréductible. Intéressons-nous d'abord au cas d'une boucle qui ne contient pas d'autres boucles.

Pour traiter une boucle, nous supposerons que son état abstrait d'entrée $d^e\prime$ est connu, nous nommons également p^e son point d'entrée. On associe tout d'abord à la boucle un "état de tête de boucle", initialisé à $d^e\prime$. On utilise l'algorithme de la figure 2.4 sur le corps de la boucle en prenant l'état de tête de boucle comme état d'entrée, jusqu'à obtenir les états abstraits sur tous les arcs retour. On fait l'union de tous les états abstraits des arcs retours ainsi que de l'état de tête de boucle, et on stocke le résultat dans l'état de tête de boucle. Ainsi, l'état de tête de boucle est désormais une abstraction valide pour tous les états concrets pouvant arriver à la tête de boucle après 0 ou 1 itération(s). On recommence

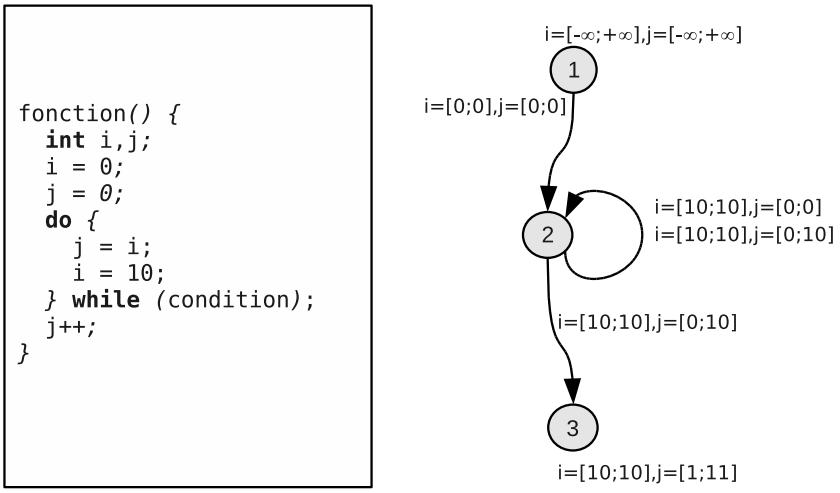


FIGURE 2.6: Interprétation abstraite et boucles

ce procédé (application de l'algorithme de la figure 2.4 et mise à jour de l'état de tête de boucle), et l'état de tête de boucle représente maintenant les états concrets possibles après 0, 1, ou 2 itérations(s). On continue jusqu'à ce qu'on atteigne un point fixe pour l'état de tête de boucle. A ce moment-là, il ne sert à rien de continuer car les résultats suivants seront toujours les mêmes. C'est pourquoi l'état de tête de boucle obtenu représente les états concrets qui peuvent arriver à la tête de boucle pour toutes les itérations. De même, les états qui ont été associés par l'algorithme aux points du programme dans la boucle, sont valables pour toutes les itérations.

Si toute boucle est réductible, on peut remplacer, dans le contexte parent (c'est-à-dire dans la boucle parente, ou bien dans le programme principal s'il n'y a pas de boucle parente) toute boucle L_j par un bloc de base bb_{Lj} dont les arcs entrants sont les arcs d'entrée de la boucle, et dont les arcs sortant sont les arcs de sortie de la boucle. La fonction d'*update* du bloc bb_{Lj} est donnée par le processus indiqué dans le paragraphe précédent. On peut ainsi appliquer l'algorithme de la figure 2.4 sur ce contexte parent. On peut donc appliquer ce principe récursivement sur le programme principal, et donc obtenir l'état abstrait en tout point du programme.

Étudions l'exemple de la figure 2.6, qui montre un programme contenant une boucle ainsi que le CFG correspondant. L'état juste avant la boucle, c'est-à-dire l'état après le bloc de base 1 indique que j et i valent 0. L'état stocké associé à la tête de boucle est donc actuellement $i = [0; 0], j = [0; 0]$.

- Itération 1 : l'état de tête de boucle est utilisé comme état de départ pour évaluer le corps de la boucle : à la fin de la première itération, j a pour valeur 0, tandis que i a pour valeur 10. L'état après le bloc de base 2 à la fin de la première itération est donc $i = [10; 10], j = [0; 0]$, cet état est fusionné avec l'état de tête de boucle, le nouvel état de tête de boucle est donc $i = [0; 10], [j = 0; 0]$.
- Itération 2 : l'état de tête de boucle, $i = [0; 10], [j = 0; 0]$, est employé comme état de départ pour évaluer le corps de la boucle, cette évaluation nous donne l'état

$i = [10; 10], j = [0; 10]$ après le bloc de base 2 en fin de deuxième itération. Cet état est fusionné avec l'état de tête de boucle, le nouvel état de tête de boucle est $i = [0; 10], j = [0; 10]$.

- Itération 3 : A partir de l'état de tête de boucle, on réalise l'évaluation du corps de la boucle, en fin d'itération 3 l'état après le bloc 2 est $i = [10; 10], j = [0; 10]$. Le nouvel état de tête de boucle est $i = [0; 10], j = [0; 10]$. On constate que ce nouvel état de tête de boucle est égal à l'ancien.

Puisqu'on a atteint un point fixe, on arrête le traitement de la boucle. Les derniers états obtenus au niveau de la boucle sont propagés sur les arcs de sortie de boucle, ainsi l'état après la boucle est l'état obtenu après le bloc 2 lors de la dernière itération, c'est-à-dire $i = [10; 10], j = [0; 10]$. On constate que ceci est correct par rapport au code du programme : après la boucle, i vaut forcément 10 (la boucle *do/while* réalise forcément au moins une itération), tandis que la valeur de j peut être 0 (si on n'a fait qu'une itération) ou bien 10 (si on a fait plus d'une itération). Enfin, l'évaluation du bloc de base 3 donne l'état $i = [10; 10], j = [1; 11]$ en fin de programme.

2.1.6.3 La convergence vers le point fixe, et le « widening »

Dans la description de l'algorithme, nous avons indiqué que, lors du traitement des boucles on réalise une recherche du point fixe de l'état d'entrée. Il est montré dans [21] que si l'espace des états abstraits ne comporte pas de chaîne strictement ascendante infinie (c'est-à-dire de chaîne infinie d'éléments de $D\prime$ de la forme $d_0\prime < d_1\prime < ...$) alors l'algorithme converge forcément vers un point fixe en un nombre fini d'étapes.

Si l'espace des états abstraits comporte des chaînes strictement ascendantes infinies, alors le procédé donné ci-dessus risque de ne plus fonctionner. Dans ce cas, à chaque fois qu'on fait l'union pour mettre à jour l'état de tête de boucle, on remplace l'opérateur d'union $\cup$ par un opérateur binaire de *« widening »* noté ∇. On doit avoir $(x \nabla y) \geq (x \cup y)$, et de plus toute séquence infinie $d_0\prime, d_1\prime, ...$ de la forme $d_1\prime = d_0\prime \nabla v_1, d_2\prime = d_1\prime \nabla v_2, ...$ (les valeurs $v_1, v_2, ...$ sont arbitraires) doit être non strictement croissante. La mise à jour de l'état de tête de boucle est bien une chaîne de la forme $d_1\prime = d_0\prime \nabla v_1, d_2\prime = d_1\prime \nabla v_2, ...$, les valeurs $v_1, v_2, ...$ étant les résultats des unions des arcs retour utilisés pour effectuer la mise à jour à la fin de chaque itération, et cette chaîne est strictement croissante (en effet, elle est forcement croissante car $(x \nabla y) \geq (x \cup y)$, et elle est strictement croissante car on s'arrête lorsqu'un point fixe est atteint), donc elle n'est pas infinie (par définition de l'opérateur de *widening*, ∇). Cet opérateur de widening permet donc d'assurer que la recherche du point fixe se termine, au prix d'une perte de précision.

2.1.7 Interprétation abstraite et dépendance au contexte

L'interprétation abstraite telle que présentée dans la section 2.1.6 comporte quelques limitations. En effet, considérons le programme décrit sur la figure 2.7. Ce programme

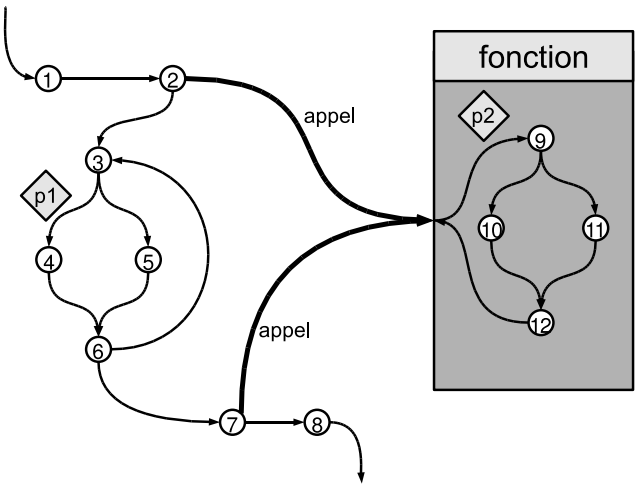


FIGURE 2.7: Interprétation abstraite et contexte

comporte une boucle dont la tête est le bloc de base 3, ainsi qu'une fonction appelée deux fois : au bloc de base 2, et au bloc de base 7. Admettons qu'on veuille faire une interprétation abstraite sur ce CFG et obtenir des états abstraits à chaque point du programme. On constate que le point *p1* est dans une boucle : l'état à ce point sera la plus petite borne supérieure sur toutes les itérations de la boucle. De même, le point *p2* est dans une fonction : l'état à ce point sera la plus petite borne supérieure sur tous les appels de la fonction. Ceci est une source d'imprécision : en effet, on ne peut pas accéder à l'état abstrait au point *p1* pour une itération donnée, ou bien à l'état *p2* pour un appel donné. Afin de pallier ce problème, il est possible de faire du déroulage de boucles, et de l'inlining de fonctions.

2.1.7.1 Déroulage de boucles

Le déroulage de boucle consiste à transformer le CFG pour dérouler les k premières itérations de chaque boucle [1], tout en conservant la sémantique du programme. Ainsi, le problème décrit sur la figure 2.7 ne se manifeste plus, car une fois la boucle déroulée, le point *p1* se retrouverait dupliqué $(k+1)$ fois (le point *p1* original, plus les k déroulages). Quand il y a plusieurs boucles imbriquées, chaque bloc est dupliqué en réalité $(k+1)^n$ fois (avec n = *degré d'imbrication*).

Un cas particulier classique est celui du déroulage de la première itération ($k = 1$). Un exemple est fourni dans la figure 2.20, ce CFG est une version déroulée de celui présenté dans la figure 2.6. La première itération de la boucle est déroulée (bloc de base 2). Les autres itérations (à partir de la deuxième) sont modélisées par la boucle comprenant le bloc de base $2'$. L'arc de retour $(2, 2)$ est dupliqué en $(2, 2')$ pour permettre de passer de la première itération aux autres itérations, et en $(2', 2')$ pour permettre de passer d'une itération autre que la première à l'itération suivante. L'arc de sortie $(2, 3)$ est dupliqué en $(2, 3)$ et $(2', 3)$ pour permettre de sortir de la boucle lors de la première itération,

1. Ce type de déroulage de boucle sera utilisé dans le présent document, toutefois il existe d'autre types de déroulage de boucle, utilisés notamment dans l'optimisation de code

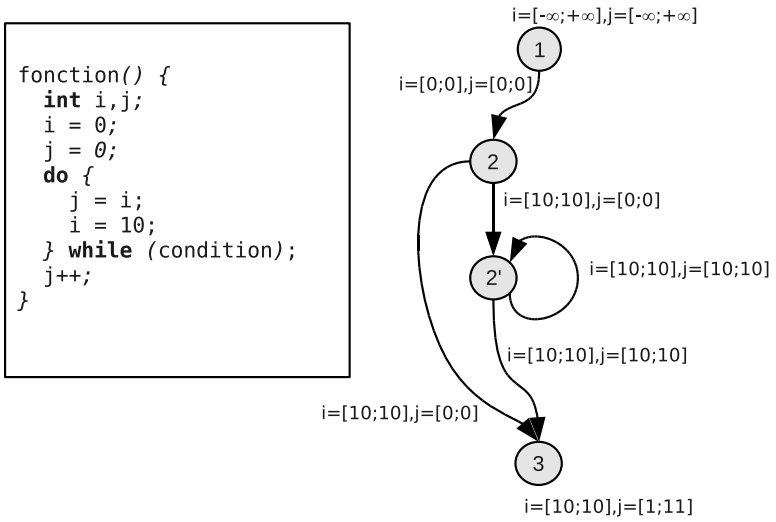


FIGURE 2.8: Déroulage de boucle

ainsi que lors d'une autre itération. Comme ceci, on peut constater que la sémantique du programme est conservée.

L'exemple est annoté par les états abstraits (intervalles) calculés à chaque arc, par l'algorithme d'interprétation abstraite appliquée aux CFGs que nous avons utilisé jusqu'ici. Supposons qu'on souhaite connaître les valeurs possibles de j juste après le bloc de base 2. Dans la version non déroulée (figure 2.20) on obtient $j = [0; 10]$, ce qui est correct dans le sens où, à ce point là du programme, la valeur de j doit être comprise entre 0 et 10. Mais si on utilise la version déroulée, on s'aperçoit qu'après le bloc 2 (correspondant au bloc de base 2, première itération), $j = [0; 0]$ et après le bloc $2'$ (correspondant au bloc de base 2, itérations suivantes) $j = [10; 10]$. Ceci nous indique que pour toutes les itérations $2..n$, la valeur de j en fin d'itération est 10. Nous verrons que ce type d'information est très utile lors de l'utilisation de l'interprétation abstraite pour la prédiction du comportement du cache d'instructions (chapitre 3).

2.1.7.2 Inlining de fonctions

L'inlining de fonctions consiste à inclure le corps des fonctions à chaque endroit où elles sont appelées. A titre d'exemple, la figure 2.9 montre la version inlinée de la fonction présente dans 2.7. Grâce à l'inlining, le point *p2* se retrouve dupliqué : on pourra accéder à l'état du programme au début de chacun des deux appels à la fonction.

De manière plus générale, avec cette méthode, pour chaque fonction il y aura prise en compte d'un contexte différent pour chaque chaîne d'appel de cette fonction. Par exemple, si la fonction g() est appelée depuis la fonction f1() et depuis la fonction f2(), alors que ces deux dernières fonctions sont appelées depuis main(), la fonction g() sera évaluée dans les contextes des chaînes d'appels (main, f1, g) et (main, f2, g). Une complication

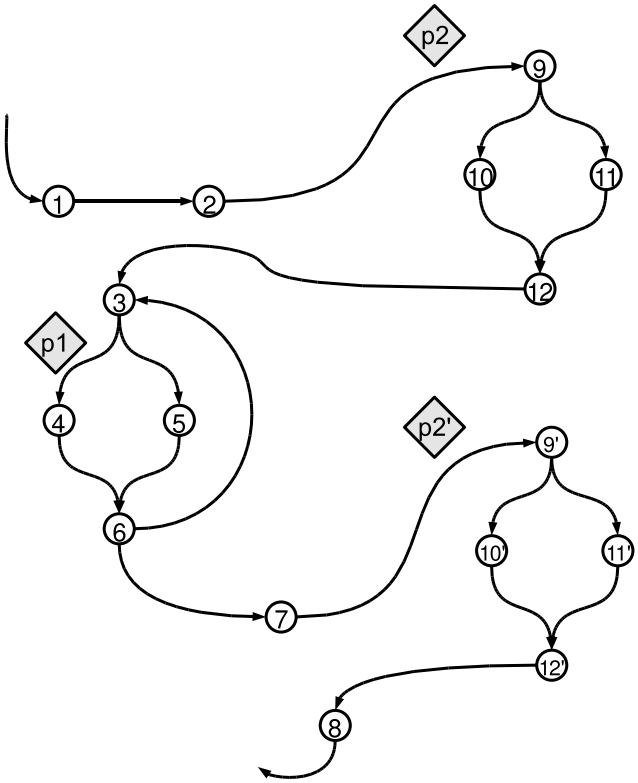


FIGURE 2.9: Inlining de fonctions

existe avec la récursivité, car elle aboutit à des chaînes d'appels dont la longueur n'est pas bornée (dans un programme correctement écrit la récursivité se termine à cause du cas d'arrêt, mais ceci est lié à la sémantique du programme, or l'inlining de fonctions n'est qu'une transformation sur la structure du CFG et donc ne s'occupe pas de la sémantique). Lorsqu'on détecte de la récursivité, c'est-à-dire lorsqu'en construisant une chaîne d'appels on s'apprête à traiter l'appel à une fonction qui est déjà dans la chaîne, alors on n'inline pas cet appel, et à la place on construit un appel vers la fonction déjà existante dans la chaîne. Ainsi, on crée une boucle dans le CFG, et cette boucle issue de la récursivité sera traitée comme une boucle habituelle, par la convergence vers un point fixe.

Un exemple de l'utilité de l'inlining des fonctions est montré grâce à la figure 2.10. Dans cet exemple, la fonction est appelée deux fois depuis le *main()*, et on cherche à obtenir les valeurs possibles de j juste avant le retour de celle-ci. Dans une analyse classique (sans inlining) la valeur de i dans l'état abstrait à l'entrée de la fonction est $i = [2; 5]$, car c'est la fusion de l'état avant chaque appel. L'analyse du corps de la fonction à partir de cet état d'entrée nous permet de constater qu'à la fin de la fonction, $j = [2; 10]$. Ceci est consistant par rapport aux valeurs concrètes possibles de j, mais cet état abstrait englobe les deux appels à la fonction.

Grâce à l'inlining, il y a deux exemplaires de la fonction, un par appel, correspondant à chaque contexte. Dans le contexte du premier appel, l'état d'entrée de la fonction est $i = [2; 2], j = [0; 0]$, l'analyse du corps de la fonction permet d'obtenir l'état $i = [10; 10], j = [2; 10]$ juste avant le retour. Lors du deuxième appel, l'état d'entrée est $i = [5; 5], j = [0; 0]$, l'analyse du corps de la fonction nous renvoie l'état $i = [10; 10]; j = [5; 10]$ à la fin de

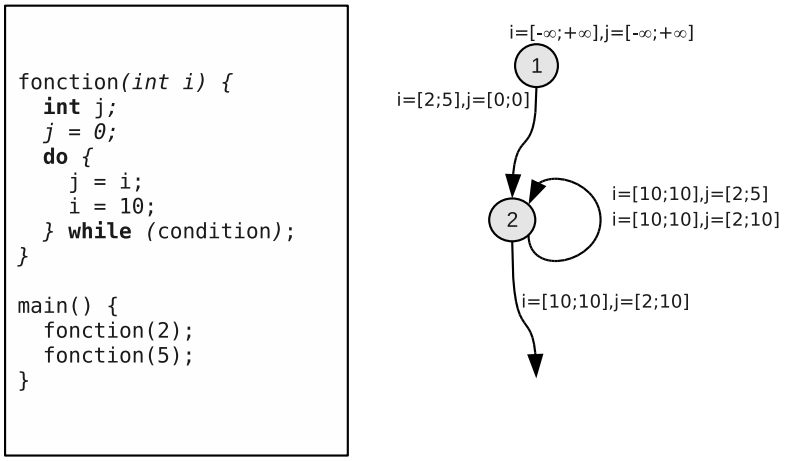


FIGURE 2.10: Exemple d'imprécision liée à une fonction appelée plusieurs fois

la fonction. Ainsi, l'inlining nous permet d'avoir les valeurs possibles de j pour chaque contexte (respectivement $j = [2; 10]$ et $j = [5; 10]$) au lieu d'avoir un état abstrait résultant de l'union de tous les contextes ($j = [2; 10]$), obtenu sans inlining, avec perte de précision.

2.2 Méthode générale du calcul de WCET

Plusieurs approches existent, mais dans tous les cas, le calcul de WCET par analyse statique est habituellement divisé en trois étapes principales.

La première étape est l'analyse du flot du programme : son but est d'identifier les chemins d'exécution qui peuvent être empruntés lors de l'exécution du programme. Pour ce faire, il est généralement nécessaire de mettre le programme sous une forme adaptée aux analyses requises (CFG, arbres, ...). Le CFG ne contient que des informations structurelles sur les chemins possibles. Or la sémantique du programme peut faire en sorte qu'un chemin qui semble possible dans le CFG ne le soit pas pas réellement (exemple : figure 2.11). L'identification de ce type de situation est effectuée dans cette étape. La détection de chemins infaisables du type de la figure 2.11 est optionnel (mais cela mène à un WCET plus précis). Toutefois la prédiction du nombre maximal d'itérations de chaque boucle est obligatoire, sinon il y aurait possibilité d'avoir un nombre infini de chemins.

Après cette étape, l'ensemble des chemins possibles est connu, mais on ne peut pas savoir lequel est le plus long, car on ne connaît pas leur temps, dépendant de l'architecture. Puisqu'on calcule un WCET, il est possible que l'analyse de flot détermine comme possibles des chemins qui ne le sont pas. Ceci provoquera du pessimisme, ce qui n'est pas problématique tant qu'il n'est pas trop important. Par contre, il faut absolument que tous les chemins possibles soit détectés.

La deuxième étape s'occupe de la modélisation d'architecture [46, 69, 80, 89] : elle prend en compte le matériel pour déterminer le temps d'exécution des chemins déterminés par l'étape précédente. Cette étape est souvent divisée en une série d'analyses, chacune prenant en compte un élément d'architecture spécifique, tel que le pipeline du processeur,

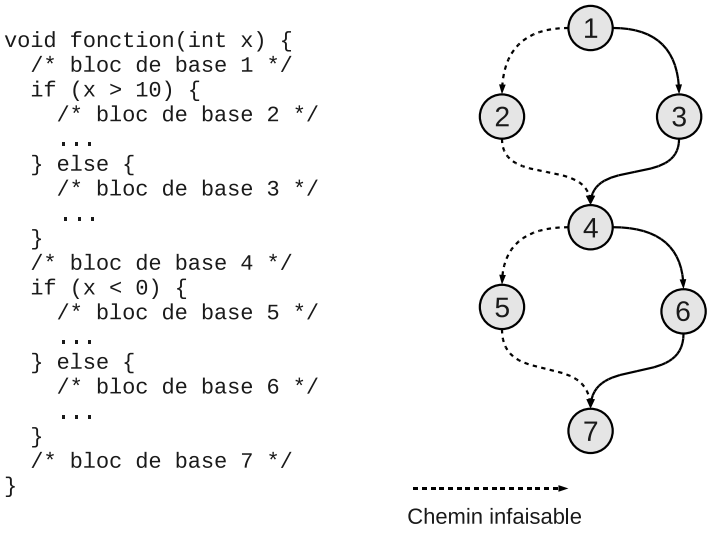


FIGURE 2.11: Exemple de chemin infaisable

la mémoire cache, le prédicteur de branchement, l'accès à la RAM, etc... Tous les éléments d'architecture doivent être pris en compte, même si c'est de manière triviale (par exemple, une manière triviale et conservatrice de modéliser le cache est de considérer qu'il n'y a que des *défauts de cache*[2]). Dans sa forme la plus simple, cette étape va simplement affecter un temps d'exécution à chaque bloc de base, qui pourra être, par exemple, obtenu par simulation.

La troisième étape est le calcul du WCET proprement dit. Il existe plusieurs approches afin de réaliser cette étape. Nous verrons trois approches couramment utilisées dans les prochaines sections : l'approche *path-based* dans la section 2.2.1, l'approche *tree-based* dans la section 2.2.2, et enfin l'approche *IPET* (*Implicit Path Enumeration Technique)* dans la section 2.2.3. C'est cette dernière approche qui sera couramment utilisée dans le reste de nos travaux, car c'est celle qui, dans l'état de l'art actuel, semble donner les résultats les plus prometteurs.

2.2.1 Techniques basées sur les chemins

A partir d'une représentation du programme sous forme de CFG, si chaque boucle est bornée alors le nombre de chemins est fini. Si on connaît le coût (temps d'exécution) de chaque bloc de base, alors on peut employer des algorithmes classiques de recherche de chemin le plus long (en fait, le plus coûteux) dans un graphe [16]. Si le chemin ainsi déterminé n'est pas possible (chemin existant dans la structure du CFG mais infaisable d'après la sémantique du programme) alors on peut l'exclure et recommencer la recherche (toutefois ce n'est pas obligatoire, si on conserve un chemin infaisable, on aura une sur-estimation du WCET, mais en aucun cas une sous-estimation). L'opération est répétée jusqu'à ce qu'on trouve un plus long chemin qui n'est pas infaisable.

2. Toutefois, l'article [61, 90] montre que le *miss* de cache ne mène pas forcément au pire temps

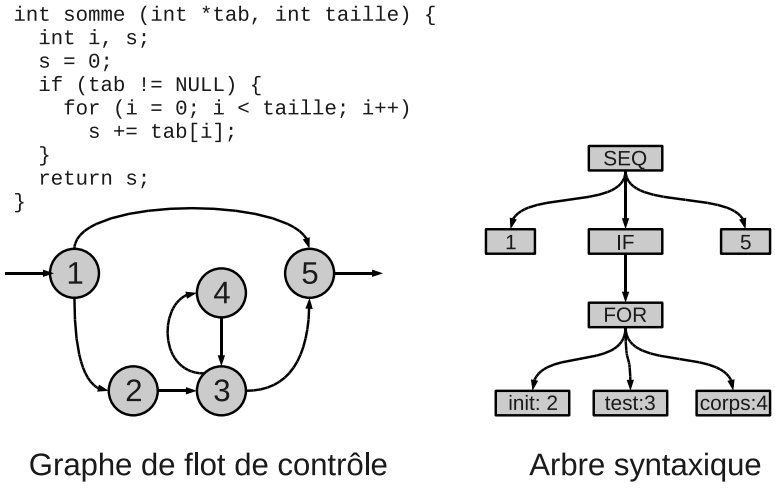


FIGURE 2.12: Différentes représentations d'un programme

2.2.2 Techniques basées sur les arbres

Les techniques *tree*-based ont été introduites dans [73]. Dans ces techniques, le programme est représenté sous forme d'arbre : chaque nœud correspond à une structure (if/then/else, for, while, do/while, appel de fonction, etc.), et les feuilles correspondent aux blocs de code. La relation de hiérarchie dans l'arbre correspond à une relation d'imbrication dans le programme. La figure 2.12 illustre un exemple de programme représenté d'abord sous forme de CFG, puis transformé en arbre.

Pour calculer le WCET, on parcourt l'arbre récursivement depuis les feuilles vers la racine : pour chaque nœud on produit un temps qui dépend des temps des nœuds fils, et ainsi de suite jusqu'à ce qu'on ait un temps pour la racine. Le système fonctionne grâce à un ensemble de règles pour chaque type de nœud (par exemple, pour un nœud de type *IF*, si on connaît les WCET des nœuds fils correspondant au THEN et au ELSE, on peut dire que le WCET du IF est le maximum des WCET du THEN et du ELSE, en plus du WCET du test).

Les temps qui sont propagés à chaque nœud peuvent être tout simplement des WCET numériques, ou bien dans le cas d'approches de type ETS (Extended Timing Schema), les résultats peuvent être des abstractions de WCET plus complexes. Plusieurs articles présentent des techniques basées sur le arbres, et proposent diverses améliorations [14, 71].

2.2.3 Technique IPET

Cette partie présente tout d'abord le principe des systèmes ILP - nécessaire à la compréhension de la méthode IPET - puis ensuite on s'intéresse à la méthode IPET proprement dite.

Problème :

$$
\begin{aligned}
x_1 &\leq 10 \\
x_2 + x_3 &= x_1 \\
x_4 &= x_2 + x_3 \\
maximiser &: 3 \times x_1 + 2 \times x_2 + 4 \times x_3 + 2 \times x_4
\end{aligned}
$$

Solution :

$$
\begin{aligned}
x_1 &= 10 \\
x_2 &= 0 \\
x_3 &= 10 \\
x_4 &= 10 \\
fonction\, objectif\, maximisee &: 3 \times 10 + 4 \times 10 + 2 \times 10 = 90
\end{aligned}
$$

FIGURE 2.13: Système ILP

2.2.3.1 Les systèmes ILP

Un système ILP (Integer Linear Programming) est constitué d'un ensemble d'équations et d'inéquations linéaires (appelées *contraintes)* portant sur des variables à valeurs entières. La *fonction objectif* est une fonction affine des variables du système, à valeur entière. Résoudre le système ILP revient à maximiser (ou minimiser) la fonction objectif, c'est-à-dire à attribuer une valeur entière à chaque variable de telle sorte que chaque contrainte soit respectée, et que le résultat de la fonction objectif soit maximal (ou minimal). Il peut n'y avoir aucune solution (si le système de contraintes est incompatible), ou bien plusieurs solutions optimales, dans ce dernier cas l'algorithme de résolution en trouvera une et il n'est pas nécessaire de les connaître toutes. La figure 2.13 montre un exemple de système ILP ainsi que sa solution. Les systèmes ILP sont utilisés dans le cadre du calcul de WCET avec la méthode IPET.

2.2.3.2 La méthode IPET

L'approche IPET a été introduite dans [53], et se base sur la représentation du programme sous forme de CFG. L'idée principale d'IPET est de représenter le flot de contrôle du programme et les effets architecturaux sous forme d'un système de contraintes linéaires. Le CFG est représenté par un ensemble de contraintes structurelles : une variable est utilisée pour représenter le nombre d'exécutions de chaque bloc de base et de chaque arc. Appelons x_i et $e_{i,j}$ respectivement les variables représentant le nombre d'exécutions du bloc de base BB_i et le nombre de passages par l'arc reliant le bloc de base BB_i au bloc de base BB_j. Pour chaque bloc de base, il existe une contrainte structurelle $\sum e_{*,i} = x_i = \sum e_{i,*}$

```
L1: Pour i allant de 1 à 10 {
 L2: Pour j allant de 1 à i {
  ...
 }
}
```

FIGURE 2.14: Exemple : *max* et *total*

(le nombre d'exécutions d'un bloc de base est égal à la somme du nombre de passages par ses arcs entrants et sortants). Si on se contente d'une analyse assez simpliste, le WCET peut alors être exprimé par la fonction objectif suivante : $\sum t_i * x_i$, où t_i représente le temps d'exécution du bloc de base BB_i, déterminé par la phase d'analyse d'effets architecturaux. En plus des contraintes structurelles, il est nécessaire d'exprimer le fait que le programme est exécuté exactement une fois par la contrainte $x_{entrée} = 1$ ($x_{entrée}$ étant le nombre d'exécutions du bloc de base d'entrée du programme). Si le programme possède des boucles, ces contraintes structurelles sont insuffisantes : il faut également représenter par des contraintes le nombre maximum d'itérations des boucles, car si le nombre maximum d'itérations des boucles n'est pas borné, le WCET ne sera pas borné. Nous décrivons dans la partie suivante l'estimation des bornes de boucles ainsi que leur représentation sous forme de contraintes ILP.

2.3 Calcul des limites de boucles

Lorsqu'on souhaite calculer le WCET d'une tâche contenant des boucles, le nombre maximum d'itérations de celles-ci doit être connu, sans quoi le WCET ne pourra pas être borné. Ces bornes (ainsi que diverses informations de flot de contrôle) peuvent être précisées par des annotations, mais la plupart du temps elles sont calculées à partir du programme à analyser [33, 28]. C'est pourquoi une étape indispensable de l'analyse de flot est le calcul des limites de boucle [38, 43, 27, 59, 23]. Nous présenterons quelques concepts communément utilisés dans cette section.

En général l'algorithme d'estimation des bornes de boucles fournit une borne *max* pour chaque boucle. L'approche de M. de Michiel et al. [24] a introduit l'utilisation d'une borne *total*, en plus de la borne *max*. La borne *max* représente le nombre maximum d'itérations de la boucle pour une entrée dans celle-ci. La borne *total* représente le maximum du nombre total d'itérations de la boucle, de manière cumulée sur un appel de la fonction contenant la boucle. Les bornes *total* et *max* ne sont bien sûr différentes que si la boucle concernée se trouve dans une autre boucle de la même fonction. En présence d'une boucle L_2 située dans une boucle L_1, la borne *total* de la boucle L_2, noté $total_{L2}$, peut toujours être approximé (par excès) par $total_{L1} \times max_{L2}$. Toutefois, il est préférable d'avoir une méthode permettant de calculer une borne *total* plus précise.

Un exemple est disponible figure 2.14. Dans cet exemple, le *max* et le *total* de la boucle L_1 sont tous deux 10. Le *max* de la boucle L_2 est 10 (ce cas est atteint lorsque i vaut 10). Le *total* de L_2 pourrait être surestimé à $total_{L1} \times max_{L2} = 100$. Toutefois la véritable

valeur de $total_{L2}$ est en fait $\sum_{i=1..10} i = 10x11/2 = 55$. Dans les sections suivantes, nous expliquerons comment les différentes approches tiennent compte de ce genre de situations.

2.3.1 Analyse des bornes de boucles par interprétation abstraite

L'approche présentée dans cette section est basée sur un travail de M. de Michiel et al. [24], qui lui-même est une amélioration d'une méthode présentée par Z. Ammarguelat et al. dans [4]. Ce travail a été mis en œvre au sein de l'outil oRange. Cette approche tente d'estimer les valeurs de *max* et de *total* pour chaque boucle du programme, en trois étapes, que nous expliciterons dans les sous-sections ci-dessous.

2.3.1.1 Étape 1

La première étape construit un arbre de contexte (*Context Tree*) pour représenter le programme. Un arbre de contexte est un arbre dont les nœuds sont les boucles ou les fonctions, et où les nœuds fils sont les boucles internes ou bien les appels de fonctions.

L'arbre de contexte est parcouru, et chaque boucle est visitée. Pour chaque boucle, on utilise une technique à base d'interprétation abstraite pour trouver quelles sont ses variables d'induction, ainsi que les incréments et les valeurs initiales de celles-ci. Les incréments peuvent être arithmétiques ($i = i + 1$) ou bien géométriques ($i = i \times 2$). A partir de la condition de boucle, et de la connaissance des variables d'induction et de leur incréments, on peut donner une expression du nombre maximum d'itérations de cette boucle. Cette expression de borne est paramétrée par le contexte de la boucle, c'est-à-dire qu'elle peut contenir des variables libres, dont les valeurs sont définies à l'extérieur de la boucle.

Par exemple, voici la boucle C suivante :

```
for (i=3; i =< n; i = i + 2) {
  somme = somme + i;
  ... reste du corps de la boucle, ne modifie pas i...
}
```

L'analyse par interprétation abstraite de cette boucle nous permettra de savoir que l'unique variable d'induction est i, que sa valeur initiale est 3, et que son incrément est de $c = 2$ (bien sûr, dans ce cas précis l'obtention de ces informations est triviale, mais dans le cas d'une boucle plus complexe qui possède plusieurs variables d'induction modifiées à différents endroits, l'interprétation abstraite devient plus que nécessaire). La condition de boucle $i < n$ permet de se rendre compte que la borne est $\left\lfloor \frac{(n-3)}{c} + 1 \right\rfloor$, et est donc paramétrée par la variable n.

A partir de ces informations, on peut obtenir une *forme normalisée* de la boucle. La boucle normalisée nécessite l'introduction d'une variable $varB$ qui sera la nouvelle variable d'induction de la boucle. Cette variable sera initialisée à 0, et incrémentée de 1 ($varB + 1$ est donc le numéro d'itération). Ainsi, la boucle normalisée sera de la forme `for (varB=0; varB < expBorne; varB++)`. La borne dans la forme normalisée est tirée

de l'expression de la borne trouvée par interprétation abstraite. Ainsi, la boucle présentée à titre d'exemple précédemment, sera représentée par la forme normalisée suivante (avec expBorne=$\left\lfloor \frac{(n-3)}{c} + 1 \right\rfloor$) :

```
for (i=3,varB=0; varB < expBorne; varB++) {
  i = i + 2;
  somme = somme + i;
  ... code de la boucle ...
}
```

Cet exemple concernait une boucle arithmétique simple. Les boucles plus complexes (boucles géométriques, variables d'induction plus complexes, conditions contenant des conjonctions, etc.) sont traitées comme expliqué dans [24].

2.3.1.2 Étape 2

Le but de l'étape 2 est d'instancier chaque expression paramétrée (une pour *max* et une pour *total*, pour chaque boucle, tel que calculé lors de l'étape précédente), ce qui permet d'obtenir les expressions définitives du *max* et du *total* pour chaque boucle. On intègre donc le contexte, et on supprime les variables libres dans ces expressions.

Pour ce faire, on utilise une interprétation abstraite dont le domaine abstrait est l'*AbstractStore* (AS). Un *AbstractStore* représente une fonction des identificateurs de variables vers des expressions symboliques de leur valeur, ces valeurs peuvent contenir des variables libres. Un AS pourra être noté, par exemple, $as_1 = [x \to 2; y \to 3, z \to k \times 6]$. On notera $as[expr]$ l'évaluation de l'expression $expr$ dans le contexte de l'AS as. L'évaluation d'une variable simple v, notée $as[v]$, retourne l'expression correspondante à v dans as si elle existe, sinon elle retourne la valeur spéciale $\top$ qui signifie qu'on ne connaît pas la valeur. L'évaluation d'une expression complexe $expr$, notée $as[expr]$ correspond à l'évaluation de l'expression $expr$ dans laquelle toutes les variables libres v_i ont été remplacées par $as[v_i]$. On définit l'opérateur de concaténation entre AS, $\circ$, de manière à ce que $\forall v, as_1, as_2$ on ait :

- $(as_1 \circ as_2)[v] = as_1[v]$ si $as_2[v] = \top$
- $(as_1 \circ as_2)[v] = as_1[as_2[v]]$ si $as_2[v] \neq \top$

La mise à jour (fonction *Update*) d'une affectation donnée sera prise en compte par l'opérateur de concaténation : $Update(as, x = y) = as \circ [x \to y]$. La jonction de chemins (fonction *Join*) sera traitée par l'union $\cup$ de telle sorte que $\forall v, as_1, as_2$ on ait :

- $(as_1 \cup as_2)[v] = as_1[v]$ si $as_1[v] = as_2[v]$ (c'est-à-dire si les expressions $as_1[v]$ et $as_2[v]$ sont égales pour tout contexte)
- $(as_1 \cup as_2)[v] = SET(exp1, exp2)$ (avec$\{as_1[v], as_2[v]\} = \{exp_1, exp_2\}$) ou bien $\top$. Le choix entre ces deux alternatives est dépendant de l'implémentation (on ne veut pas créer d'ensembles de SET imbriqués trop grands, ce qui pénaliserait la vitesse d'analyse, mais on ne veut pas non plus passer trop rapidement à $\top$ ce qui sacrifierait la précision). Dans l'implémentation de référence (oRange), les SET sont conservés pendant l'analyse d'une boucle, mais éliminés lorsqu'on sort d'une boucle

```
k = 10
m = 4
B1: for (i=0; i < k; i += 2) {
  B2: for () {
    ...corps de la boucle ...
  }
  B3: for () {
    ...corps de la boucle ...
  }
}
m = m + 2
B4: for (j=m ; j < 10; j++) {
  B5: for () {
    ...corps de la boucle ...
  }
}
```

FIGURE 2.15: Nids de boucles

externe (contenue dans aucune autre boucle), sauf pour les sets ne contenant que des constantes, qui sont eux conservés indéfiniment.

De plus, l'union d'expressions $exp1$ et $exp2$ contenant uniquement des constantes ou des SET de constantes peut être simplifiée en $SET(min, max)$ où min et max sont respectivement les constantes minimum et maximum contenue dans $exp1$ ou $exp2$. Par exemple, l'union de $SET(1, 5)$ et $SET(3, 8)$ serait $SET(1, 8)$.

A partir de ces définitions, nous savons comment traiter les séquences d'affectations, ainsi que les conditions. Intéressons-nous maintenant aux fonctions *Update* des boucles. On identifie d'abord les nids de boucles. Un nid de boucles est un ensemble de boucles N, tel qu'il existe une boucle $b_1 \in N$ appelée tête de nid et que pour toute boucle $b_i \in N$, $b_1$inclut b_i, et que $b_1$ne soit inclus par aucune boucle. Ainsi, chaque nid de boucles peut être considéré comme un arbre, dont chaque nœud représente une boucle, et dont la racine représente la boucle tête de nid. Une boucle b_i représente un nœud fils (dans l'arbre) de la boucle b_j si la boucle b_j contient la boucle b_i, et qu'il n'existe pas de boucle b_k tel que b_j contient b_k et b_k contient b_i. Les feuilles correspondent aux boucles ne contenant pas d'autre boucles.

Pour traiter un nid de boucle, on doit parcourir ses boucles de la plus interne à la plus externe (la tête de nid, donc), de manière à ce que l'on ne traite une boucle que lorsque ses sous-boucles internes ont été traitées également, ceci peut être réalisé par un parcours en profondeur d'abord de l'arbre représentant le nid de boucles. Définissons tout d'abord le traitement d'une boucle interne ne contenant pas d'autres boucles. Ensuite, nous définirons la manière de traiter une boucle contenant d'autre boucles dès lors que ces boucles internes auront été traitées.

Pour traiter une boucle interne b_i (ne contenant pas d'autres boucles), on réalise l'interprétation abstraite avec les *AbstractStore* sur le corps de la boucle normalisée, jusqu'à obtenir un point fixe. Soit $AS_2 = UpdateCorps(AS_1)$ la fonction *Update* correspondant

au corps de la boucle. On applique $UpdateCorps()$ sur la boucle, et ensuite on examine l'*AbstractStore* obtenu à la fin du corps de la boucle, et pour chaque variable, on vérifie si celle-ci peut être ré-écrite en fonction de $varB$, la variable d'induction de la boucle normalisée. Par exemple pour un morceau d'*AbstractStore* $[x \rightarrow x + K]$ on ré-écrit en $[x \rightarrow x_0 + K \times (varB + 1)]$ (x_0 étant la valeur de x avant la boucle). L'*AbstractStore* résultat est modifié de telle sorte qu'on remplace toute occurrence de $varB$ par $(varB - 1)$. L'*AbstractStore* ainsi modifié est joint à l'état d'entrée de la boucle, et utilisé comme état d'entrée de l'itération suivante de l'interprétation abstraite, comme expliqué dans la section 2.1.6 sur l'interprétation abstraite liée aux CFG. Une fois le point fixe atteint, on obtient un *AbstractStore* correspondant aux variables modifiées dans la boucle, qui est ensuite enrichi par deux affectations, $maxB_i = expBorne_i$, et $totalB_i = expBorne_i$. Les valeurs de $maxB_i$ et $totalB_i$ sont récupérées à partir des informations calculées lors de l'étape 1. La boucle est ensuite annotée par l'*AbstractStore* ainsi calculé.

Lorsqu'une boucle b_e contenant d'autre boucles internes doit être traitée, alors les boucles internes b_i ($i = [1..n]$) doivent être traitées au préalable, c'est-à-dire qu'elles doivent être annotées avec des *AbstractStore* contenant les affectations à $maxB_i$, $totalB_i$ ainsi qu'aux autres variables modifiées dans ces boucles. Les expressions des $maxB_i$ et $totalB_i$ des boucles internes sont modifiées pour les ramener au contexte de la boucle courante : soit $varB$ le compteur d'itération de la variable courante, si $maxB_i = expMaxB_i$ et $totalB_i = expTotalB_i$, alors ces expressions deviennent respectivement $maxB_i = MAX(varB = 0..expBorne_e, expMaxB_i)$ et $totalB_i = \sum_{varB=0}^{expBorne_e} expTotalB_i$ (nous rappelons que $expBorne_i$ est l'expression de borne de la boucle i calculée par l'étape 1). Une fois ceci réalisé, on peut réaliser l'interprétation abstraite de la boucle courante de la même manière que présenté plus haut pour les boucles internes, et l'*Update* correspondant à une boucle interne est défini par la composition avec l'*AbstractStore* stocké dans l'annotation de la boucle interne.

Ce processus est effectué jusqu'à ce qu'on ait traité la tête de nid. Une fois ceci fait, on a obtenu les $maxB_i$ et $totalB_i$ de chaque boucle, dans le contexte de la tête de nid. L'*Update* du nid est donc défini par la composition avec l'*AbstractStore* stocké dans l'annotation de la boucle externe.

L'interprétation abstraite définie précédemment est appliquée sur le programme (à partir d'un contexte d'entrée modélisée par une AS tenant compte des paramètres d'appel du programme, des valeurs initiales des variables globales, etc.). Pendant cette interprétation abstraite, la mise à jour (*Update*) de chaque nid de boucle entraîne le calcul des limites des boucles de ce nid.

Pour produire les expressions de *max* et *total* définitions pour chaque boucle, il ne reste plus qu'une opération à réaliser : pour chaque occurrence d'une variable de type $varB_i$ dans les expressions, on remplace par $SET(0, expMaxB_i)$, où $expMaxB_i$ est l'expression du max de la boucle B_i (boucle correspondante à la variable $varB_i$). Le SET est requis car l'expression du max n'est qu'une estimation, le max réel peut être inférieur ou même nul.

2.3.1.3 Étape 3

L'étape précédente a produit les *max* et *total* définitifs. Toutefois, ceux-ci restent encore sous la forme d'expression. Cette dernière étape évalue les expressions, nous permettant d'obtenir des valeurs numériques. L'analyse retourne donc l'arbre de contexte annoté par les valeurs du *max* et du *total* pour chaque boucle. Si cette dernière phase n'arrive pas à calculer la valeur numérique d'une expression (par exemple en cas de variable libre), alors on peut aussi retourner l'expression non évaluée.

2.3.2 Approche de S. Bydge et al. pour l'estimation des bornes de boucles

Dans l'article [10], S. Bydge et al. proposent une approche de calcul de WCET paramétrique, qui inclut, entre autre, une méthode qui permet de borner le nombre de passages par chaque bloc de base du programme (dans cette section, nous nous limiterons à parler de cet aspect de l'article de S. Bydge et al., toutefois ce papier n'est pas du tout restreint à ce sujet-là). Ces bornes peuvent donc nous permettre d'obtenir le nombre total d'itérations de chaque boucle.

Dans cet article, le programme est noté P et l'ensemble de ses blocs de base est noté Q_P. L'ensemble des variables du programme est noté V_P, et l'ensemble des paramètres d'entrée est noté I_P (on a $I_P \subset V_P$ car les paramètres d'entrée sont des variables du programme). Les bornes de passage par les blocs de base Q_P sont exprimés paramétriquement en fonction des paramètres d'entrées I_P, et calculés grâce à la fonction $MEC_P : \mathbb{Z}^{|I_P|} \to \mathbb{N}^{|Q_P|}$.

L'idée de base utilisée pour réaliser ce calcul est la suivante : le programme P est déterministe, son état est entièrement décrit par la valeur des variables de l'ensemble V_P. Ainsi, si on prend pour hypothèse que le programme se termine, alors pour un point du programme donné, chaque état $E \in \mathbb{N}^{|V_P|}$ doit être unique (car si on repasse deux fois par le même état, étant donné que le programme est déterministe, on retrouvera cet état un nombre infini de fois). Donc, pour un point du programme donné, le nombre d'exécutions est borné par le nombre d'états possibles à ce point.

L'interprétation abstraite permet de prédire l'ensemble des états qui seront possibles durant l'exécution (ou du moins, un ensemble incluant l'ensemble des états possibles). Il est ensuite possible de compter ces états possibles, et donc de borner le nombre d'exécutions des blocs de base. Cette idée pourrait s'adapter à plusieurs types d'interprétation abstraite, mais l'article de S. Bygde et al. a choisi d'utiliser une interprétation abstraite dont le domaine est un ensemble d'inéquations linéaires qui définissent les valeurs possibles des variables à chaque point du programme. Ainsi, l'ensemble des valeurs des variables possibles à un point donné du programme peut être vu comme l'intérieur d'un polyèdre convexe dans un espace à $|V_P|$ dimensions, délimité par les inéquations linéaires. Compter le nombre d'états possibles revient à compter les solutions au système d'inéquations à chaque point, et une méthode pour faire cela de manière symbolique a été proposée par

W. Pugh dans [72]. Cette méthode a été reprise et simplifiée par S. Bygde et al.

2.3.3 Approche de N. Holsti et al. pour l'estimation des bornes de boucles

Dans l'article [49], N. Holsti présente une méthode assez innovante pour traiter le problème des chemins infaisables lors du calcul de WCET. Sa méthode permet également d'analyser les limites de boucles. Dans son approche, N. Holsti réalise une analyse des valeurs en modélisant le domaine (l'ensemble des valeurs possibles des variables) par un tuple d'entiers soumis à une série de contraintes linéaires (*ensembles de Presburger*). L'analyse associe ainsi un *Ensemble de Presburge*r à chaque point du programme. Le WCET est exprimé sous forme d'une variable supplémentaire dans le domaine, ce qui fait qu'il n'y a plus besoin d'une étape spécifique (de type IPET) pour calculer le WCET : l'analyse des valeurs va calculer la valeur de la variable représentant le WCET.

Pour calculer les limites de boucles, dans chaque boucle l'analyse des valeurs est enrichie par des variables d'incréments, une pour chaque variable qui est modifiée dans la boucle (par exemple une variable x intervenant dans la boucle se verra associer la variable d'incrément dx). Après le calcul, on connaît les valeurs possibles de chaque incrément. Pour chaque variable, si on obtient un incrément borné, la variable est une *variable d'induction*. Si l'incrément est nul, la variable est *invariante*. Si l'incrément n'est pas borné, alors on n'a pas d'informations sur le comportement de la variable (cas appelé *fuzzy variable* dans l'article original).

Une fois les variables catégorisées et les incréments connus, on introduit une nouvelle variable c représentant le numéro d'itération de la boucle, et on lie, par des contraintes, cette variable aux invariants et variables d'induction de la boucle (par exemple, si x est une variable d'induction et que x_0 est sa valeur initiale, on aura $x = x_0 + c \times dx$). Ensuite, l'examen des conditions de sortie permet de borner c et donc de connaître les bornes du nombre d'itérations de la boucle.

2.4 Analyse du cache

Le cache est une mémoire rapide et limitée en taille, utilisée pour accélérer l'accès à la mémoire principale. Des copies des blocs de la mémoire principales sont stockés dans le cache, ce qui permet au processeur d'y accéder plus rapidement. Quand un accès à la mémoire a lieu, soit la donnée demandée est présente dans le cache et la récupération de la donnée est rapide (cas appelé *cache hit*), soit elle n'est pas présente et il faut prendre le temps de la récupérer dans la mémoire principale (cas appelé *cache miss)*.

Le cache est divisé en lignes de cache de taille fixe, et la mémoire principale est divisée en blocs de cache. Chaque bloc de cache en mémoire correspond à une ligne de cache spécifique. En cas de *miss*, tout le bloc de cache contenant l'instruction ou la donnée concernée est chargé dans la ligne de cache correspondante. Dans un cache associatif par

ensemble de niveau A, chaque ensemble peut contenir jusqu'à A blocs. Par conséquent, une politique de remplacement est nécessaire pour déterminer où mettre les blocs à charger dans le cache et quel bloc supprimer du cache. Une politique de remplacement couramment utilisée est LRU (*Least Recently Used*). Cette politique associe un âge $a \in [0, A[$ à chaque bloc dans le cache. Quand on tente d'accéder à un bloc qui n'est pas dans le cache, ce dernier est chargé tout en écrasant le bloc le plus vieux qui s'y trouvait. Le bloc nouvellement chargé dans le cache se voit affecter l'âge 0, et l'âge de tout les autres blocs dans la même ligne est incrémenté. Quand on accède à un bloc qui est déjà dans le cache, ce bloc est rafraîchi (c'est à dire que son âge est mis à 0), et tous les blocs de la ligne dont l'âge était inférieur au bloc accédé voient leur âge augmenter. Aucun bloc n'est expulsé du cache dans ce cas-là.

Il existe d'autres politiques de remplacement : on peut citer la politique MRU (*Most Recently Used*) qui remplace le bloc le plus récemment utilisé (à l'inverse de LRU), la politique FIFO qui se comporte comme LRU excepté qu'un bloc n'est jamais rafraîchi, ainsi que la politique Pseudo-LRU qui est une approximation de LRU beaucoup moins coûteuse à implémenter au niveau du matériel. Dans l'article [78], J. Reineke et al. présentent une étude qui compare les différentes politiques de remplacement en terme de prédictibilité (plus une politique de remplacement est prédictible, plus elle est propice au calcul de WCET). Pour chaque politique de remplacement, l'article présente deux métriques, $fill(A)$ et $evict(A)$, qui permettent d'associer à A (le niveau d'associativité du cache) un entier qui permet d'estimer la prédictibilité de cette politique de remplacement. Ces entiers estiment, à partir d'un état de cache inconnu, le nombre de références nécessaires pour converger vers un état déterminé : plus ces entiers sont importants, moins la politique de remplacement est prédictible. L'étude présentée par l'article [78] montre que le cas le plus déterministe est le LRU, où les métriques $fill(A)$ et $evict(A)$ croissent linéairement en fonction de A. Pour la politique FIFO, $evict(A)$ et $fill(A)$ croissent aussi linéairement mais avec une pente plus élevée. La politique MRU présente des cas où $fill(A)$ est infini, et avec la politique Pseudo-LRU la croissance de $evict(A)$ et $fill(A)$ est super-linéaire par rapport à A. Les politiques autres que LRU (et à la rigueur FIFO, mais la politique FIFO n'offre pas des performances très intéressantes en terme de rapidité d'exécution) sont donc peu adaptées au calcul de WCET. Toutefois, le Pseudo-LRU est assez couramment utilisé dans le matériel (PowerPC 75X, Intel Pentium II-IV). Heureusement, un article différent de J. Reineke et al. [77] fait référence au fait qu'il est possible, pour prendre en compte un cache Pseudo-LRU, d'approximer sa politique de remplacement par du LRU d'un niveau d'associativité moindre, et ce de manière conservatrice. C'est pourquoi nous nous intéresserons principalement à l'étude des caches LRU.

Grâce au cache d'instruction, la récupération des instructions de la tâche à analyser est parfois rapide (en cas de succès de cache), parfois plus lente (en cas de défaut de cache). Ceci a un effet sur le WCET, c'est pourquoi il est souhaitable de pouvoir prédire le comportement du cache afin d'avoir un WCET le plus précis possible.

Pour modéliser le cache d'instruction dans le cadre du calcul du WCET, plusieurs

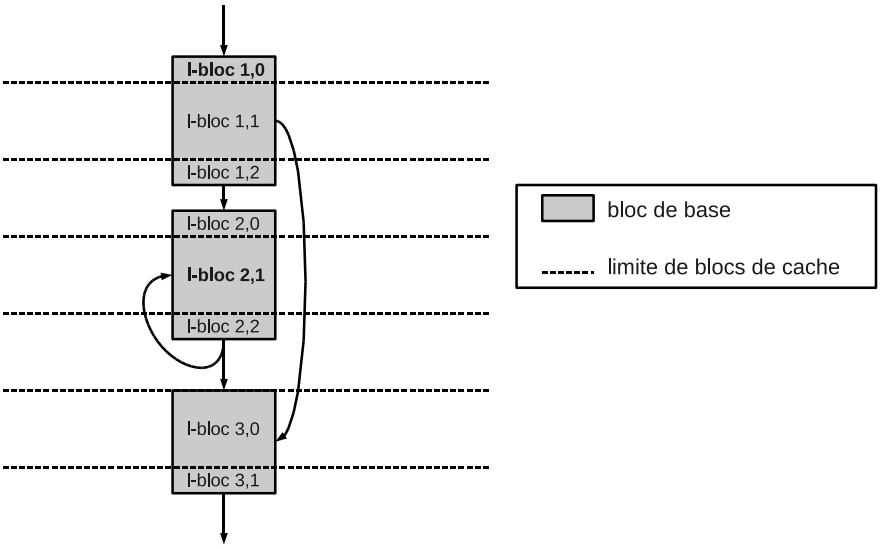


FIGURE 2.16: Les l-blocs

approches ont été proposées [57, 95], chaque approche possède ses avantages et ses inconvénients. Nous présenterons quelques approches communément utilisées dans cette sous-section.

Les analyses de cache d'instruction utilisent souvent le concept de *l-bloc.* Un l-bloc peut être défini comme le plus grand bloc de code contigu appartenant au même BB et au même bloc de cache (en d'autres mots, projeté sur la même ligne). Ainsi, un BB qui est réparti sur n blocs de cache sera partitionné en n l-blocs (figure 2.16). Le bloc de cache d'un l-bloc lb donné sera noté par convention $bloc_{lb}$, et la ligne ou ensemble sur lequel ce bloc est projeté sera noté $ligne_{lb}$. On pourra aussi se référer à un l-bloc par une paire (b, l) dans laquelle b se réfère au numéro du bloc de base dans lequel est le l-bloc, et l désigne l'index (en partant de 0) du l-bloc dans le bloc de base. Ainsi, par exemple, le 4ème l-bloc du bloc de base 2 sera désigné par la paire $(2, 3)$.

Étant donné que les blocs de cache n'interagissent qu'avec d'autres blocs de cache de la même ligne, un l-bloc n'aura pas d'influence sur les l-blocs qui ne sont pas dans sa ligne. C'est pourquoi les analyses de cache, en général, regroupent les l-blocs par ligne de cache, et réalisent une analyse indépendante pour chaque ligne, ce qui permet de réduire le temps de calcul et la consommation de mémoire.

2.4.1 Les graphes de conflit de cache (CCG) de Li et al.

Dans l'article [54] de Li. et al., la technique du CCG (Cache Conflict Graph) est présentée. Cette méthode est utilisée pour modéliser le cache d'instructions. Cette technique consiste à créer un CCG pour chaque ligne du cache. Le CCG est la projection du CFG sur la ligne de cache. Ses nœuds sont les l-blocs du programme qui projetés sur cette ligne (plus deux nœuds spéciaux pour l'entrée et la sortie), et ses arcs sont dérivés du CFG : un l-bloc *lb2* (appartenant au BB nommé *bb2*) est successeur de *lb1* (appartenant à *bb1)*

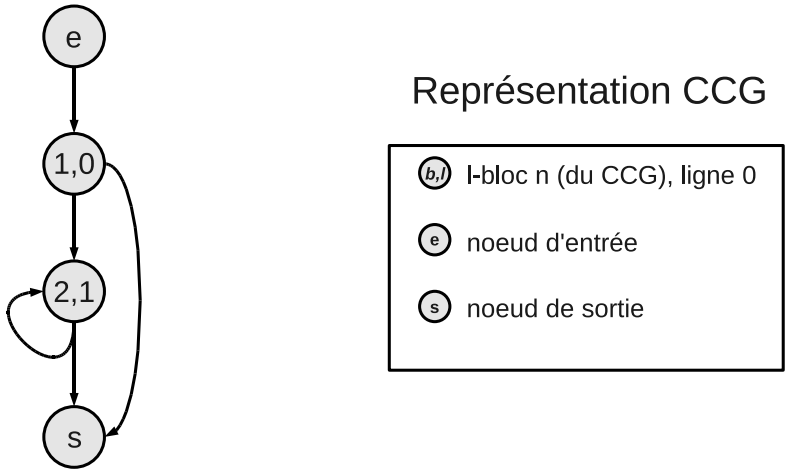


FIGURE 2.17: Le CCG

si et seulement si il existe un chemin dans le CFG allant de *bb1* vers *bb2* sans passer par un BB contenant un autre l-bloc de la même ligne. A titre d'exemple, le CCG est montré sur la figure 2.17, ce CCG correspond au CFG montré dans la figure 2.16 (les l-blocs en gras sont ceux appartenant à la ligne de cache utilisée pour créer le CCG).

Des variables $x_{i,j}$ sont créées pour chaque l-bloc de chaque CCG, représentant le nombre d'exécutions du l-bloc. Des variables $p_{(i1,j1)\rightarrow(i2,j2)}$ sont également construites pour exprimer le nombre d'exécutions des arcs du CCG, et des contraintes structurelles similaires à celles du CFG sont générées. De plus, pour tout i, j une contrainte $x_{i,j} = x_i$ permet d'exprimer le fait que le nombre d'exécutions d'un l-bloc est égal au nombre d'exécutions du BB dans lequel il est inclus.

On rajoute des variables de la forme $x_{i,j}^{miss}$ et $x_{i,j}^{hit}$ permettant de comptabiliser le nombre d'exécutions du j-ième l-bloc du BB i dans le cas du défaut et du succès de cache (on a bien sur $x_{i,j}^{miss} + x_{i,j}^{hit} = x_{i,i}$ pour chaque l-bloc i, j). Les temps de chaque l-bloc sont représentés par des variables de la forme $t_{i,j}^{miss}$ (pour les défauts de cache) et $t_{i,j}^{hit}$ (pour les succès de cache). La fonction objectif à maximiser devient donc $\sum x_{i,j}^{miss}.t_{i,j}^{miss} + x_{i,j}^{hit}.t_{i,j}^{hit}$. Ensuite, la structure même du CCG permet de rajouter des contraintes ILP visant à limiter le nombre de défauts de cache estimés. Par exemple, tel que montré sur la figure 2.18, lorsqu'on un arc du CCG fait une boucle, cela signifie qu'il existe un chemin dans le CFG partant d'un l-bloc donné et aboutissant à ce même l-bloc sans passer par un autre l-bloc de la même ligne, par conséquent le bloc de cache n'aura pas pu être remplacé, et il y aura automatiquement un succès de cache. Donc on aura une contrainte de type $x_{i,j}^{hit} \geq p_{(i,j)\rightarrow(i,j)}$ pour exprimer ce fait. Ce n'est qu'un exemple de ce qu'on peut faire, une liste exhaustive se trouve dans [54]. Une fois les contraintes générées on peut maximiser la fonction objectif, le WCET calculé tiendra compte du cache.

Cette technique permet une modélisation très précise du cache et s'intègre bien à IPET, mais son point faible est le nombre de contraintes générées. En particulier, la prise en compte du cache associatif par ensemble au moyen du CCG [55] engendre un nombre

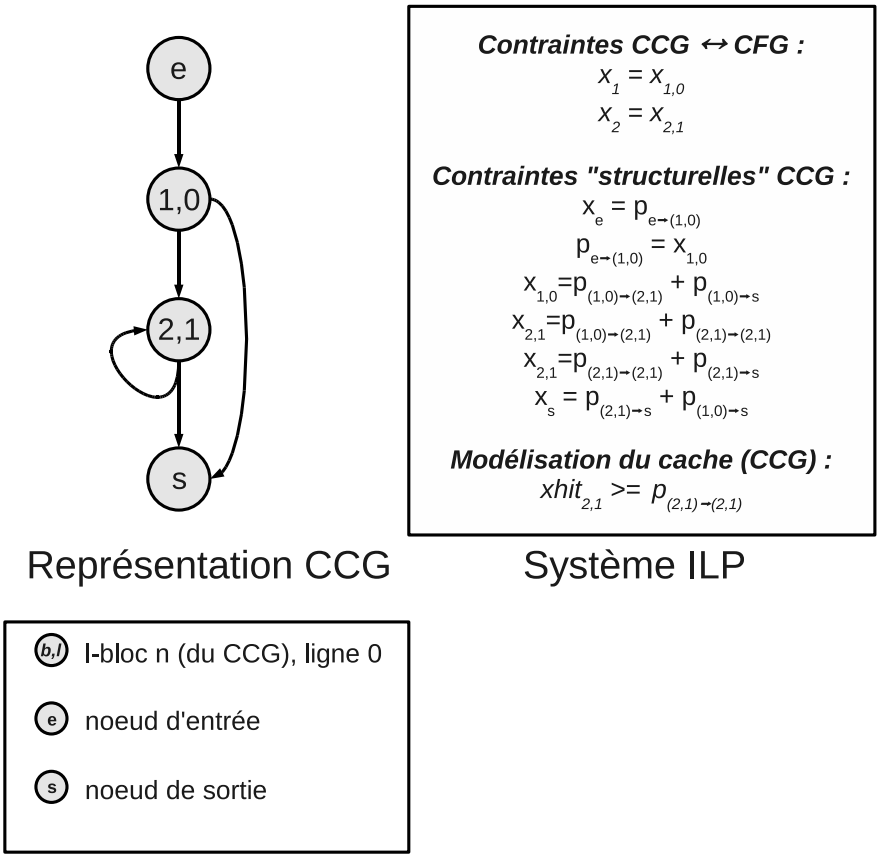


FIGURE 2.18: Utilisation du CCG pour générer des contraintes ILP

prohibitif de contraintes (croissant de manière exponentielle par rapport au nombre de voies du cache). En effet, le grand nombre de contraintes provoque de gros délais lors de la résolution du système ILP.

2.4.2 La simulation statique du cache d'instructions de Mueller et al.

La simulation statique du cache, présentée pour la première fois par F. Mueller et al. dans [67], est une technique d'analyse statique du comportement du cache d'instructions.

Cette technique était initialement prévue pour les caches d'instructions à application directe. On traite chaque ligne de cache séparément. La simulation statique du cache d'instructions à application directe revient à faire une analyse de type DFA, comme présenté dans la section 2.1.4, pour chaque ligne.

Pour chaque ligne courante notée lc, l'ensemble utilisé pour le DFA est l'ensemble des blocs de cache de cette ligne, augmenté d'un bloc spécial I représentant le bloc *invalide.* Cet ensemble augmenté sera noté E. L'ensemble d'entrée du DFA est égal au singleton $\{I\}$, ceci modélise le fait qu'au début du programme on ne sait pas ce qui est dans le cache d'instructions. Nous noterons $ligne_{inst}$ la ligne de cache contenant l'instruction $inst$, et nous noterons aussi $bloc_{inst}$ le bloc de cache contenant $inst$. La jonction entre chemins est

modélisée par l'union d'ensembles classiques, et les ensembles GEN et $KILL$ sont définis de la manière suivante pour chaque bloc de base :

- Soit $LI = \{inst : ligne_{inst} = lc\}$. Si $LI = \emptyset$ alors $GEN = \emptyset$. Sinon, soit $inst$ la dernière (selon l'adresse mémoire) instruction de LI, on a $GEN = \{bloc_{inst}\}$.
- Soit $LI = \{inst : ligne_{inst} = lc\}$. Si $LI = \emptyset$ alors $KILL = \emptyset$. Sinon, soit $inst$ la dernière (selon l'adresse mémoire) instruction de LI, on a $KILL = \{E - bloc_{inst}\}$.

L'analyse de DFA est ensuite lancée pour obtenir la valeur des ensembles en tout point du programme. Nous noterons ens_{inst} l'ensemble qui sera juste avant l'instruction I (en fait l'analyse DFA calcule les ensembles avant et après chaque bloc de base, mais il est trivial d'obtenir les ensembles avant chaque instruction d'un bloc de base en fonction de l'ensemble à l'entrée de celui-ci).

L'analyse de Mueller utilise alors ces ensembles calculés par le DFA pour classer chaque instruction dans une de ces quatre catégories :

- *Always Hit* (AH), pour les instructions dont l'exécution provoquera toujours un *hit* de cache. Une instruction *inst* possédera cette catégorie si et seulement si $ens_{inst} = \{bloc_{inst}\}$. En d'autre mots, pour que l'instruction ait une chance de causer un *hit* il faut d'une part que l'ensemble contienne $bloc_{inst}$ (ce qui indique que le bloc peut se trouver dans le cache au moment de l'exécution de l'instruction), et il faut aussi que l'ensemble ne contienne pas d'autre blocs (ce qui indique qu'aucun autre bloc que $bloc_{inst}$ ne pourrait se trouver dans le cache, donc cela implique que $bloc_{inst}$ sera forcément dans le cache).
- *Always Miss* (AM), pour les instructions dont l'exécution provoquera toujours un *miss* de cache. Une instruction *inst* possédera cette catégorie si et seulement si $bloc_{inst} \notin ens_{inst}$. Dans ce cas, ens_{inst} permet de constater que $bloc_{inst}$ ne peut pas se trouver dans le cache au moment de l'exécution de *inst*. Par conséquent, l'instruction provoque forcément un *miss* de cache.
- *First Miss* (FM), pour les instructions dont la première exécution provoquera soit un *hit* soit un *miss*, mais dont les exécutions suivantes provoqueront forcément des *hits*. Ce cas se produit lorsque $bloc_{inst} \in ens_{inst}$ (l'instruction peut être un *hit*), mais $\exists bloc2 : (bloc2 \in ens_{inst}) \wedge (bloc2 \neq bloc_{inst})$ (l'instruction n'est pas forcément un *hit* car il y a d'autre blocs qui pourraient être dans le cache lors de l'exécution de l'instruction). A ce moment-là, pour montrer que *inst* ne peut causer que des *hits* une fois que la première exécution est passée, il faut montrer que pour chaque bloc $bloc2$ vérifiant la condition $(bloc2 \in ens_{inst}) \wedge (bloc2 \neq bloc_{inst})$, ce $bloc2$ ne peut pas être dans le cache au moment de l'exécution *inst*, du moment que cette dernière a déjà été exécutée au moins une fois. Pour cela, on vérifie qu'il n'y a pas de chemin dans le CFG qui part de *inst* pour arriver vers *inst* en passant par une instruction contenue dans $bloc2$. Ainsi, on est sûr qu'une fois qu'on a exécuté *inst* une fois, on ne passera jamais par une instruction provoquant l'éviction de $bloc_{inst}$ du cache, pour ensuite revenir sur *inst*.
- *Conflict* (C), pour les instructions dont le comportement vis-à-vis du cache est plus

compliqué, et qui ne pourront donc pas être modélisées par l'une des catégories ci-dessus. Toute instruction n'étant pas catégorisée *Always Miss, First Miss,* ou bien *Always Hit* sera par défaut catégorisée *Conflict*.

Ensuite, cette catégorisation permet la prise en compte du cache dans le WCET, par exemple en générant des contraintes IPET supplémentaires. La technique de Mueller et al. a donc été prévue initialement pour le cache d'instructions à application directe, mais cette technique a été plus tard étendue au cache associatif par ensembles dans l'article [66], ainsi qu'aux multiples niveaux de cache dans l'article [65].

2.4.3 L'interprétation abstraite du cache de Ferdinand et al.

Dans différents articles [34, 87, 3], C. Ferdinand, M. Alt et al. proposent une méthode pour gérer le cache d'instructions par catégorisation, avec IPET. Cette méthode classe les instructions en trois catégories : *Always Hit (AH)* pour les instructions dont l'exécution provoquera toujours un *hit*, et *Always Miss (AM)* pour les instructions dont l'exécution provoquera toujours un *miss*. Enfin, la catégorie *Not Classified (NC)* est utilisée pour les instructions sur lesquelles on ne peut pas faire de prédiction sûre.

Les catégories sont calculées par interprétation abstraite sur le CFG du programme. Des États Abstraits de Cache (*ACS - Abstract Cache State*) sont calculés avant et après chaque BB. Ces ACS servent ensuite à déterminer les catégories car ils représentent un ensemble d'états concrets de cache qui peuvent potentiellement exister pendant l'exécution réelle du programme. Les ACS sont calculés par interprétation abstraite sur le CFG à partir de fonctions *update* et *join* (comme expliqué dans la partie sur l'interprétation abstraite).

Deux analyses différentes sont réalisées : l'analyse *May* et l'analyse *Must*. Bien que similaires, chaque analyse utilise son propre type d'ACS. Chaque type d'ACS associe un ensemble s de blocs de cache à chaque ligne de cache, et est noté $ACS^p_{type}(l)$ (où l est la ligne de cache, où *type* est *may* ou *must*, et où p est un point du programme comme par exemple un bloc de base ou un l-bloc). Un âge noté a est associé à chaque bloc. Dans l'analyse *May* (resp. *Must*), l'ensemble s contient l'ensemble des blocs qui peuvent (resp. qui doivent) être dans le cache, et l'âge $a \in [0, A[$ représente une borne inférieure (resp. supérieure) de leur âge réel. L'ensemble des blocs de cache de la ligne l d'âge a qui sont dans l'ACS à un point du programme p est noté $ACS^p_{type}(l, a)$. L'âge d'un bloc de cache donné *cb* pourra être représenté par $ACS^p_{type}[cb]$. Dans ce cas, par convention nous noterons que $ACS^p_{type}[cb] = A$ si le bloc *cb* n'est pas dans l'ensemble $ACS^p_{type}(l)$ (où l est la ligne de cache de *cb*). Des fonctions *Update* et *Join* distinctes existent pour chaque analyse. Ces fonctions sont détaillées dans la table 2.19. Les fonctions *Update* définies dans le tableau prennent en entrée une ACS ainsi qu'un l-bloc, et retournent l'ACS après être passé par le l-bloc. Pour obtenir une fonction $Update_{BB}$ travaillant un bloc de base et non un l-bloc, il suffit de composer les fonctions *Update* sur tout les l-blocs contenus dans le bloc de base. Le point du programme p pourra être, entre autres, un bloc de base pour désigner l'ACS obtenue par l'analyse avant ce bloc de base (exemple : $ACS^{BB}_{type}(l)$)

Condition	Catégorie
$\exists a \neq A/bloc_{lb} \in ACS^{lb}_{must}(ligne_{lb}, a)$	Always Hit (AH)
$\forall a \neq A/bloc_{lb} \notin ACS^{lb}_{may}(ligne_{lb}, a)$	Always Miss (AM)
dans tous les autres cas	Not Classified (NC)

TABLE 2.1: Règles de catégorisation pour le l-bloc lb

ou bien un l-bloc pour désigner l'ACS avant ce l-bloc (exemple : $ACS^{lb}_{type}(l)$). La valeur spéciale $\perp$ est définie de telle manière que $\forall x$, $Join(\perp, x) = x$, tandis que $\top$ est défini de telle manière que $\forall x$, $Join(\top, x) = \top$. La valeur $\emptyset$ représente l'ACS dans lequel aucun bloc de cache n'est présent. Les autres notations possibles seront expliquées au fur et à mesure que cela sera nécessaire.

A la fin de l'analyse, les ACS sont utilisés pour calculer les catégories, comme expliqué dans la table 2.1. Ici, nous considérons un CFG projeté sur les blocs de cache (c'est-à-dire que chaque bloc de base est découpé selon les limites de blocs de cache). Ainsi, seule la première instruction de chaque bloc de base peut faire l'objet d'une prédiction non triviale (les autres instructions provoqueront forcement des *hits*). C'est pourquoi, pour les besoins de l'explication, on pourra affecter une catégorie par bloc de base.

Les catégories AH, AM, et NC sont insuffisantes pour obtenir un WCET précis. En effet, on peut imaginer un programme comportant une boucle dans lequel un bloc provoquera un *miss* la première itération, et des *hits* automatiques pour toutes les autres itérations. A ce point de l'analyse, ce bloc serait classifié en tant que NC, sans tirer parti du fait que toutes les itérations autres que la première provoqueront *hits*. Ce type de l-bloc est communément appelé bloc *First Miss*. Il existe plusieurs raffinements de l'analyse permettant de tenir compte de ce type de situation, nous présentons deux approches importantes ci-dessous.

2.4.3.1 Le déroulage de boucles (loop unrolling)

La première approche que nous allons présenter est le déroulage de boucles, qui est décrit (entre autres) dans un article assez complet de F. Martin et al. [63]. Le déroulage de boucles déroule la première itération de chaque boucle (schéma 2.20]) avant de réaliser les analyses *May* et *Must*. Par conséquent, chaque bloc qui est dans une boucle est évalué dans deux contextes : le contexte *"première itération"* et le contexte *"autres itérations"*. Quand il y a plusieurs boucles imbriquées, chaque bloc est dupliqué en réalité 2^n fois (avec n = *degré d'imbrication*).

Grâce à cette technique, un bloc qui ne cause que des *hits* à partir de la deuxième itération pourra être catégorisé AH pour le contexte *"autres itérations"* et donc le WCET sera plus précis. Le déroulage de boucles doit se faire au début dans le processus de calcul de WCET, afin que tous les autres traitements (en particulier la génération des contraintes structurelles liées au CFG) se fassent sur la version déroulée du CFG.

$Update_{must}(ACS, lbloc)(l, a) = e(a)$, *tel que* :

$$\begin{array}{rcl} si \quad \exists h/ & & lbloc \in e(h) : \\ e(0) & = & \{lbloc\}, \\ e(i) & = & ACS(l, i-1)/i = [1..h[\\ e(h) & = & ACS(l, h-1) \cup ACS(l, h) - lbloc \\ e(i) & = & ACS(l, i)/i \in]h..A[\\ si \quad \forall h & & lbloc \notin e(h) : \\ e(0) & = & \{lbloc\}, \\ e(i) & = & ACS(l, i-1)/i \in [1..A[\end{array}$$

$Update_{may}(ACS, lbloc)(l, a) = e(a)$, *tel que* :

$$\begin{array}{rcl} si \quad \exists h/ & & lbloc \in e(h) : \\ e(0) & = & \{lb\}, \\ e(i) & = & ACS(l, i-1)/i = [1..h] \\ e(h+1) & = & ACS(l, h+1) \cup ACS(l, h) - lbloc \\ e(i) & = & ACS(l, i)/i \in]h+1..A[\\ si \quad \forall h & & lbloc \notin e(h) : \\ e(0) & = & \{lbloc\}, \\ e(i) & = & ACS(l, i-1)/i \in [1..A[\end{array}$$

$\forall l, a \; Join_{must}(ACS_1, ACS_2)(l, a) =$

$$\{b : \exists a_1, a_2/b \in ACS_1(l, a_1),\, b \in ACS_2(l, a_2),\, a = max(a_1, a_2)$$

$\forall l, a \; Join_{may}(ACS_1, ACS_2)(l, a) =$

$$\begin{array}{rcl} \{b : \exists a_1, a_2/b & \in & ACS_1(l, a_1),\, b \in ACS_2(l, a_2),\, a = min(a_1, a_2)\} \cup \\ \{b : b & \in & ACS_1(l, a),\, \forall a'\, b \notin ACS_2(l, a')\} \cup \\ \{b : b & \in & ACS_2(l, a),\, \forall a'\, b \notin ACS_1(l, a')\} \end{array}$$

FIGURE 2.19: Interprétation abstraite pour le cache

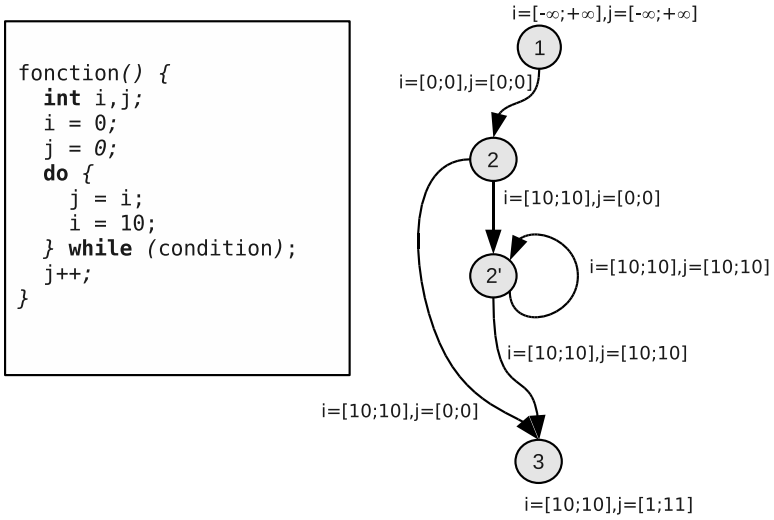


FIGURE 2.20: Déroulage de boucle

2.4.3.2 La *persistence*

La deuxième approche est l'analyse de *persistence*, présentée par C. Ferdinand et al. dans [32]. Cette approche rajoute une nouvelle analyse en plus du *May* et du *Must,* appelée analyse de *persistence* (notée *Pers),* utilisant sa propre sorte d'ACS. Le but de cette analyse est de trouver les blocs dits *persistents*, c'est-à-dire ceux qui ne peuvent pas être éliminés du cache une fois qu'ils ont été chargés. Ces blocs causeront au pire un *miss* pour tout le programme. Comme pour les ACS *May* et *Must*, l'ACS de l'analyse de *persistence* associe un ensemble s de blocs de cache à chaque ensemble de cache. Un âge $a \in [0..A[\cup l_\top$ est associé à chaque bloc, et représente une borne supérieure de leur âge réel (l'âge $l_\top$ est considéré comme le plus vieux de tous, pour les besoins de la comparaison) . Le nouvel âge (virtuel) noté $l_\top$ est affecté aux blocs qui peuvent avoir été mis dans le cache puis enlevés. Une nouvelle catégorie *Persistent* (PS) est définie : un l-bloc sera considéré *Persistent* si et seulement si $\exists block_B \in ACS_B^{pers}(l, a) \wedge a \neq l_\top$.

Grâce à cette nouvelle analyse, les blocs dans une boucle pour lesquels il n'y a que des *hits* à partir de la deuxième itération seront catégorisés PS, ce qui permettra d'avoir un WCET plus précis.

2.4.3.3 Intégration avec IPET

Les résultats de l'analyse de cache doivent être intégrés dans la méthode IPET. Pour ce faire, on crée les variables x_i^{hit} et x_i^{miss} représentant respectivement le nombre de *hit* et de *miss* pour le bloc de base i. Dans tous les cas, on aura $x_i^{hit} + x_i^{miss} = x_i$ pour tout bloc de base i. Des contraintes supplémentaires peuvent être générées suivant la catégorie de i. Par exemple, si la catégorie est *Always Miss*, on aura une contrainte $x_i^{miss} = x_i$ (car tous les accès au bloc seront des *miss*). La table 2.2 indique les contraintes supplémentaires générées en fonction de chaque catégorie.

Catégorie	Contrainte
Always Hit	$x_{i,j}^{miss} = 0$
Persistent	$x_{i,j}^{miss} \leq 1$
Always Miss	$x_{i,j}^{miss} = x_i$
Not Classified	$x_{i,j}^{miss} \leq x_i$

TABLE 2.2: Génération de contraintes

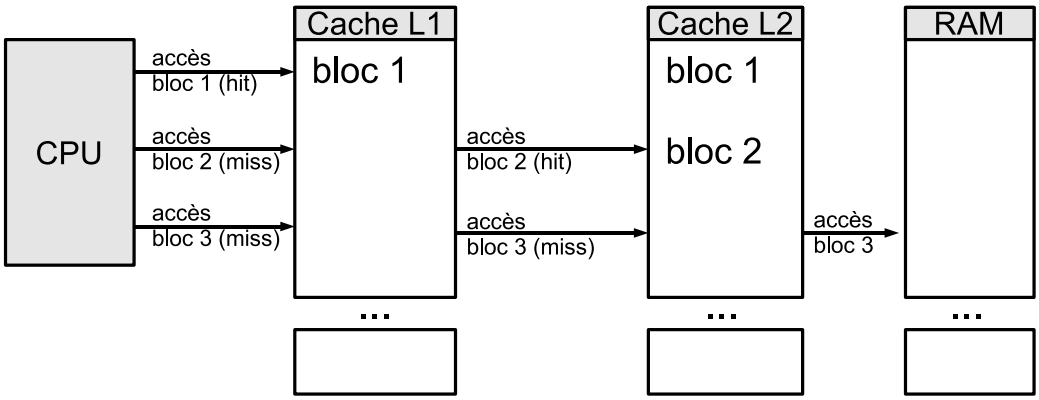


FIGURE 2.21: Caches multi-niveaux

2.4.3.4 Prise en compte de multiples niveaux de cache

Jusqu'ici, nous avons vu uniquement des hiérarchies mémoires ne comprenant qu'un seul cache d'instructions, et de la mémoire. Il est possible d'avoir une hiérarchie mémoire constituée de plusieurs caches. Lorsque le processeur doit charger une nouvelle instruction, une requête est faite pour le cache d'instructions de niveau 1. Si ce cache ne peut pas satisfaire la requête (cache *miss*), alors le cache niveau 1 fait une requête au cache niveau 2, et ainsi de suite jusqu'à ce qu'un cache puisse satisfaire la requête, ou bien que la requête arrive jusqu'à la mémoire centrale (RAM). Ce processus est illustré dans la figure 2.21.

Il est montré dans l'article [41] que les techniques d'analyse de cache d'instructions par catégorisation peuvent s'adapter, de manière générale, à une hiérarchie comprenant n caches d'instructions, en plus de la mémoire centrale. Le cache de niveau 1 peut être catégorisé par la méthode d'interprétation abstraite du cache de C. Ferdinand et al. Toutefois, chaque analyse de niveau supérieur à 1 est dépendante des résultats de l'analyse du niveau de cache précédent. En effet, si une requête est satisfaite (*hit*) pour un niveau donné $L \in [1..n[$, cette requête ne sera pas envoyée au niveau $i + 1$.

Pour prendre ceci en compte, on catégorise chaque référence r à une instruction à l'aide de deux ensembles de catégories, et ce pour chaque niveau de cache :

– La classification $CHMC_{r,L}$ (pour *Cache Hit Miss Classification*) représente la catégorisation "habituelle" des instructions (par exemple : *Always Hit*, *Always Miss*, *Persistent*, *Not Classified*). S'il y a un accès au cache de niveau L pour la référence

r, alors cette classification permet de prédire le type d'accès (*hit* ou *miss*), mais elle ne permet pas de prédire si il y aura effectivement un accès au cache de niveau L.

- La classification $CAC_{r,L}$ (pour *Cache Usage Classification*) permet de savoir s'il y aura un accès au cache de niveau L ou pas, pour la référence r. Les catégories disponibles sont A (*Always* - obligatoirement un accès au cache), N (*Never* - jamais un accès au cache), et U (*Unknown* - il y aura peut-être un accès au cache).

Pour le cache de niveau 1, on réalise la classification $CHMC$ comme on le ferait pour un système de cache d'instructions à un seul niveau avec la méthode d'interprétation abstraite de C. Ferdinand et al. La classification CAC pour le cache de niveau 1 associe la catégorie A pour toutes les références. En effet, lorsqu'une référence se produit, on va forcément accéder au cache de niveau 1. Ensuite, on peut facilement voir que la catégorie CAC d'une référence pour le niveau L dépend de la catégorie $CHMC$ et de la catégorie CAC du niveau $L-1$. En effet, le cache niveau L sera accédé à condition que le cache niveau $L-1$ l'ait été, et que cet accès ait été un *miss*. En pratique, cela veut dire que $CAC_{r,L} = A$ si $(CAC_{r,L-1} = A) \wedge (CHMC_{r,L-1} = AM)$, et que $CAC_{r,L} = N$ si $(CAC_{r,L-1} = N) \vee (CHMC_{r,L-1} = AH)$. Dans tous les autres cas, on aura $CAC_{r,L} = U$.

La fonction *Update* de l'analyse par interprétation abstraite des niveaux de cache supérieur à 1 est légèrement modifiée comme suit : puisque lorsqu'on réalise la fonction *Update* d'une référence r pour un niveau de cache L, on n'est pas sûr que cet accès soit réellement effectué, on consulte la catégorie $CAC_{r,L}$. Si la catégorie est A, alors rien ne change par rapport à une fonction *Update* traditionnelle. Si la catégorie est N alors le cache n'est pas accédé, et la fonction *Update* est l'identité (comme s'il n'y avait pas eu la référence). Dans le cas où la catégorie est U, on réalise l'union (*Join*) des deux cas pour prendre en compte les deux possibilités.

Ensuite, du point de vue de la prise en compte de l'analyse dans le WCET, chaque référence r doit faire l'objet d'une pénalité liée à chaque niveau de cache L à condition que cette référence provoque l'accès au cache de niveau L et que cet accès soit un *miss*. Ceci peut être facilement implémenté à l'aide de contraintes IPET, de manière similaire à ce qui est réalisé pour une hiérarchie mémoire constituée d'un seul niveau de cache.

2.4.3.5 Extension aux caches de données

Jusqu'ici nous n'avons travaillé qu'avec des caches d'instructions. Il peut également être intéressant de prendre en compte les caches de données [92, 93, 81, 76, 51]. La principale difficulté dans la gestion des caches de données vient du fait que pour une référence donnée (c'est-à-dire en général une instruction de type LOAD ou STORE), on ne connaît pas l'adresse mémoire, et donc on ne peut pas savoir quel est le bloc de cache concerné.

Par conséquent, la première chose à envisager lors d'une analyse de cache de données est l'affectation d'un ensemble d'adresses possibles pour chaque instruction qui accède à la mémoire. Quelques méthodes d'analyse d'adresses sont données dans les articles [40] et [36]. On voit aisément que les variables globales sont les cas les plus simples à prendre en compte. Le cas des accès à des variables de pile est un peu plus compliqué mais possible,

l'accès au tas est plus difficile à prendre en compte.

Pour gérer le cas de la pile, il faut d'abord réaliser une analyse de pile. Cette analyse de pile a pour but de fournir, pour chaque fonction, l'ensemble des différentes adresses de base possibles du cadre du pile (l'adresse indiquée par le registre pointeur de pile lors de l'entrée dans la fonction). Pour ce faire, on peut utiliser une approche de type DFA ou interprétation abstraite pour calculer la valeur du registre pointeur de pile en tout point du programme. Une abstraction doit être fournie, un exemple pourrait être l'ensemble des adresses. Un *Join* (une possibilité est l'union des ensembles d'adresses, avec passage à $\top$ lorsqu'il y a un nombre infini d'adresses possibles à cause par exemple de la récursivité) et un *Update* (qui pourrait être aussi simple que la mise à jour des adresses de l'ensemble lorsqu'on trouve une instruction qui déplace le pointeur de pile). Ces éléments doivent être choisis pour s'adapter au niveau de précision requis. Si on veut une adresse de cadre de pile précise pour chaque fonction, on peut employer l'inlining. Il faut simplement faire attention aux fonctions récursives (voir section 2.1.7.2).

Une fois cette analyse de pile effectuée, il est possible d'associer à chaque référence mémoire une liste d'adresses possibles (où bien l'élément $\top$ pour dire qu'on ne sait pas). A partir de là, on peut réaliser une analyse du cache par interprétation abstraite comme expliqué, par exemple, dans la section 2.4.3 (bien que ce ne soit pas la seule façon possible de faire, on pourrait par exemple aussi utiliser la méthode de simulation statique du cache). La fonction *Update* d'un bloc de base contenant des instructions d'accès mémoire devra, pour chacune de ces instructions, procéder à l'injection du ou des blocs de cache référencés dans l'ACS. En cas d'instruction pouvant accéder à plusieurs blocs mémoires, la fonction *Update* devra prendre la borne supérieure (*Join* des ACS) de toutes les possibilités.

Une fois ceci fait, il sera possible de catégoriser les accès au cache de données et de tenir compte de ces catégories dans l'approche IPET, de manière similaire à ce qui est décrit dans la section 2.4.3.

2.5 Modélisation du Pipeline

Grâce à l'utilisation du pipeline, le temps d'exécution d'une séquence d'instructions est inférieur au temps d'exécution de chaque instruction isolée. Il en est de même pour l'exécution en séquence de plusieurs blocs de base. La prise en compte des effets de pipeline permet d'aboutir à un WCET plus précis. Souvent, on distingue deux formes d'analyse de pipeline : (1) l'analyse intra-bloc qui s'occupe de l'effet du pipeline sur les instructions à l'intérieur d'un bloc et (2) l'analyse inter-blocs qui prends en compte les effets de recouvrement entre plusieurs blocs de base.

Il existe différentes techniques de prise en compte du pipeline pour le calcul de WCET [56, 44, 25], nous en présenterons un certain nombre dans cette section.

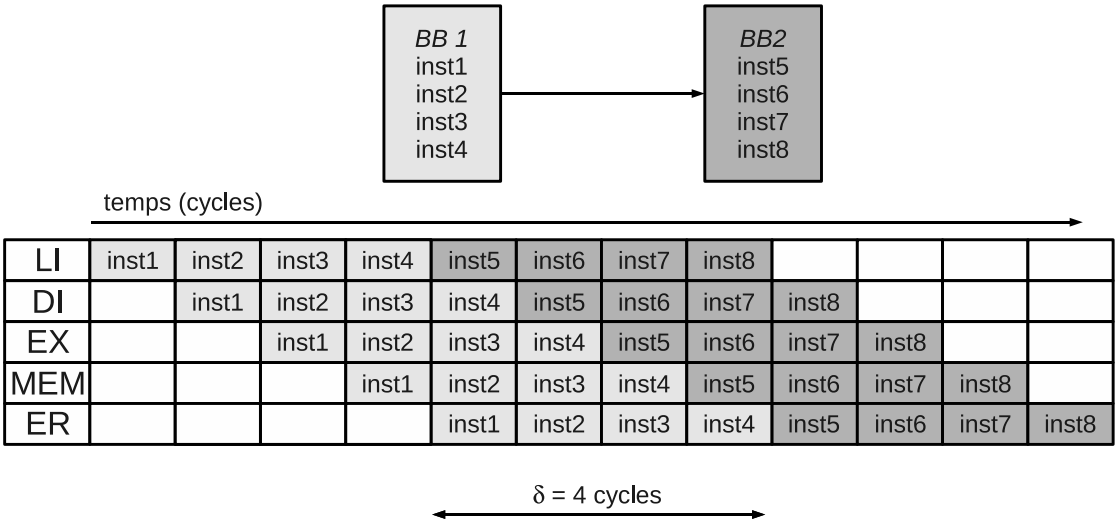


FIGURE 2.22: Recouvrement de blocs

2.5.1 La technique des *deltas* de J. Engblom et al.

Dans l'article [26], J. Engblom et al. ont proposé une méthode de prise en compte du pipeline pour le calcul de WCET. Cette méthode est une technique d'analyse de pipeline inter-blocs. Elle fonctionne en associant à chaque séquence de deux blocs de base (ou plus) un temps négatif (noté *δ)*. Ce δ représente le gain de temps lié au recouvrement des blocs de la séquence (figure 2.22). La longueur maximale (en nombre de blocs de base) des séquences prises en compte est un paramètre de l'analyse

Le gain de temps associé à chaque séquence peut être obtenu, par exemple, par l'utilisation d'un simulateur. Ensuite, ces informations sont prises en compte au niveau du système ILP : pour chaque séquence s une variable ILP x_{seq} est créée pour représenter le nombre d'exécutions de cette séquence et un terme $\delta_s \times x_{seq}$ est rajouté à la fonction objectif. Des contraintes ILP adéquates sont ensuite générées pour borner x_{seq}. Par exemple, si la séquence s est composée des deux blocs B_1 et B_2, alors on aura $x_{seq} = e_{1,2}$ (où $e_{1,2}$ représente le nombre de passages par l'arc du bloc B_1 au bloc B_2).

Pour une séquence de deux prédécesseurs (donc de trois blocs au total), l'expression du nombre d'exécutions de cette séquence est plus difficile, mais néanmoins réalisable. Soit (B_1, B_2, B_3) une telle séquence, et soit x_{seq} la variable ILP représentant le nombre d'exécutions de cette séquence. Pour garder la même convention, exprimons également par $e_{i,j}$ le nombre de passages par l'arc (B_i, B_j). D'une part, il est évident qu'on a $x_{seq} \leq e_{1,2}$ et $x_{seq} \leq e_{2,3}$, car chaque passage par la séquence constitue aussi un passage par $e_{1,2}$ et $e_{2,3}$. Ceci nous permet de placer une borne supérieure sur le nombre d'exécutions de la séquence.

Essayons maintenant de trouver une borne inférieure. Soient $B_{s1}, ..., B_{sn}$ les n successeurs de B_2, en excluant B_3, et soient $B_{p1}, ..., B_{pm}$ les m prédécesseurs de B_2, en excluant B_1. Soient $(x_{s1}, ..., x_{sn})$ les n variables ILP correspondant aux n séquences représentées par l'expression ci-dessous :

$$((B_1, B_2, B_{s1}), ..., (B_1, B_2, B_{sn})).$$

De la même manière, définissons $(x_{p1}, ..., x_{pm})$ pour les séquences représentées par l'expression ci-dessous :

$$((B_{p1}, B_2, B_3), ..., (B_{pm}, B_2, B_3)).$$

Il est facile de constater que $\sum_{i=1}^{n} x_{si} + x_{seq} \geq e_{1,2}$ car chaque emprunt de l'arc $e_{1,2}$ implique qu'on va emprunter une séquence commençant par B_1 et finissant par B_3 ou bien un des $(B_{s1}, ..., B_{sn})$, et ce à cause des contraintes structurelles du CFG. Pour les même raisons, on a $\sum_{i=1}^{m} x_{pi} + x_{seq} \geq e_{2,3}$ On peut donc rajouter des contraintes de type $x_{seq} \geq e_{1,2} - \sum_{i=1}^{n} x_{si}$ et $x_{seq} \geq e_{2,3} - \sum_{i=1}^{m} x_{pi}$ pour exprimer la borne inférieure de x_{seq}.

La méthode pour traiter les séquences de longueur supérieure suit le même principe, des précisions sont fournies dans l'article [26]. Au final, la résolution du système ILP ainsi enrichi permet d'obtenir le WCET en tenant compte du gain de temps lié au recouvrement de blocs grâce au pipeline.

2.5.2 L'approche par interprétation abstraite de J. Schneider et al.

Dans l'article [35], J. Schneider et C. Ferdinand proposent une méthode de prise en compte des effets de pipeline dans le WCET. Cette méthode utilise l'interprétation abstraite, et supporte les processeurs superscalaires, avec exécution dans l'ordre ou dans le désordre.

Comme la plupart des méthodes basées sur une interprétation abstraite sur le CFG, celle-ci calcule des états de pipeline abstraits à chaque point du CFG. Pour définir cette interprétation abstraite, nous avons au moins besoin de définir le domaine abstrait (état de pipeline abstrait), ainsi que les fonctions *Update* et *Join*.

Définissons d'abord la sémantique concrète (état de pipeline concret et fonction *Update* concrète) avant de nous intéresser à la sémantique abstraite. Le pipeline est constitué d'un ensemble d'étages noté $PS = \{LI, DI, EX, ...\}$ (pour *Pipeline Stages*). De plus, le processeur comporte des ressources qui peuvent être demandées avant d'accepter une instruction donnée dans un étage de pipeline donné. Appelons $R = \{r_1, ..., r_n\}$ l'ensemble des types de ressources et des ressources du processeur. On appelle une *association de ressource* une paire $(s, \{r_1, ..., r_n\})$ avec $s \in PS$ et $r_1, ..., r_n \in R$.

Une *séquence d'associations de ressource* est notée $\bar{r} \in \bar{R} = R^*$, et l'opérateur "." est défini comme étant la concaténation de ces séquences. Une *séquence de demande de ressources* est une séquence d'associations de ressource décrivant les besoins en ressources d'une instruction donnée. Ces besoins dépendent du type d'instruction ainsi que de sa forme (opérandes mémoire ou registre, par exemple), ceci est prédéfini statiquement pour chaque instruction du jeu d'instructions du processeur, et dépend de l'architecture utilisée.

Une *séquence d'allocation de ressource* est une séquence d'associations de ressource qui décrit l'affectation des ressources à des instructions qui se trouvent à un instant donné dans un étage du pipeline.

L'état concret du pipeline est donc défini par $p = (r\#, s_R)$, où $r\# \in \bar{R}^{N \times |PS|}$ représente l'ensemble des séquence d'allocation de ressource (au maximum une par instruction se trouvant dans le pipeline, N est le nombre maximum d'instructions pouvant être dans le même étage en même temps, $N > 1$ dans le cas d'un processeur superscalaire), et où s_R modélise l'état d'autres ressources diverses du processeur n'appartenant pas à R (ceci dépend du processeur modélisé, mais par exemple l'article [35] utilise s_R pour modéliser l'état de la *Prefetch Queue*). Soit P l'ensemble des états concrets possibles du pipeline.

Soit IS le jeu d'instructions du processeur que l'on modélise. La fonction *Update* concrète est de type $P \times IS \rightarrow P$, c'est-à-dire qu'à partir d'un état concret de pipeline et d'une instruction, elle retourne l'état concret du pipeline une fois que l'instruction rentre dans celui-ci. Son implémentation est spécifique au processeur modélisé, mais un exemple pour le processeur *SuperSPARC I* est donné dans l'article [35].

La fonction *Cycles,* notée C et de type $P \times IS \rightarrow \mathbb{N}_0$ renvoie le nombre de cycles nécessaires pour que l'instruction passée en paramètre rentre dans le pipeline. L'implémentation de cette fonction est spécifique au processeur. De manière similaire, la fonction *Empty*, notée E et de type $P \rightarrow \mathbb{N}_0$ renvoie le temps (en nombre de cycles), à partir d'un état concret de pipeline donné, qui est nécessaire pour que toutes les instructions contenues dans le pipeline soient complètement traitées. Cette fonction est également spécifique au processeur modélisé.

La sémantique abstraite est définie de manière assez simple : tout d'abord, l'état de pipeline abstrait est défini comme étant un ensemble d'états de pipeline concrets. C'est-à-dire que $\hat{P}$, l'ensemble des états de pipeline abstraits, est égal à 2^P. A partir de ceci, il est simple d'étendre les fonctions *Update*, *Cycles*, et *Empty* pour que celles-ci travaillent sur les états de pipeline abstraits. Ainsi, la fonction *Update* abstraite notée $\hat{U}$ est définie de telle manière que $\hat{U}(\hat{p}, i) = \{U(p, i) p \in \hat{p}\}$. La fonction *Cycles* abstraite notée $\hat{C}$ peut être étendue de telle manière que $\hat{C}(\hat{p}, i) = max(p \in \hat{p}, C(p, i))$. De la même manière, la fonction *Empty* abstraite notée $\hat{E}$ peut être définie par $\hat{E}(\hat{p}) = max(p \in \hat{p}, E(p))$.

Quant à la fonction *Join* utilisée lorsque deux chemins dans le CFG se rencontrent, elle est simplement définie comme état l'union des ensembles d'états concrets de pipeline, c'est-à-dire que $p \in \hat{J}(\hat{p}1, \hat{p}2) \iff p \in \hat{p}1 \vee p \in \hat{p}2$. Toutes les fonctions qui travaillent sur des instructions isolées peuvent être étendues pour pouvoir travailler sur des blocs de base. Par exemple pour la fonction *Update*, si un bloc de base BB est composé d'une série d'instructions $i_1, i_2, ..., i_n$, alors $\hat{U}(\hat{p}, BB) = \hat{U}(...\hat{U}(\hat{U}(\hat{p}, i_1), i_2)..., i_n)$. De même, la fonction *Cycles* peut être étendue par $\hat{C}(\hat{p}, BB) = \hat{C}(\hat{p}, i_1) + \hat{C}(\hat{U}(\hat{p}, i_1), i_2) + ... + \hat{C}(...\hat{U}(\hat{U}(\hat{p}, i_1), i_2)..., i_n)$.

Une fois muni de ces fonctions, on peut réaliser l'interprétation abstraite du pipeline sur le CFG du programme comme présenté dans la section 2.7 (comme d'habitude, il est aussi possible de réaliser du déroulage de boucle ou de l'inlining de fonctions pour améliorer la précision). Le résultat sera un état de pipeline abstrait, calculé en tout point du CFG. A partir de ces états, on peut obtenir des informations temporelles sur les effets du pipeline à l'aide des fonctions *Cycles* et *Empty*.

Notons $\hat{p}_{BB}^{entrée}$ l'état de pipeline abstrait avant le bloc de base BB, tel que trouvé par

l'interprétation abstraite. Notons également $\hat{p}_{BB}^{sortie}$ l'état en sortie. Pour un bloc de base BB donné, on peut lui affecter un coût d'exécution $\hat{C}(\hat{p}_{BB}^{entrée}, BB)$, représentant le temps nécessaire pour que les instructions de BB rentrent dans le pipeline. De plus, si le bloc de base BB est le bloc de sortie du CFG, il faut rajouter au coût la valeur de $\hat{E}(\hat{p}_{BB}^{sortie})$ à son coût pour prendre en compte le temps de vidage du pipeline à la fin du programme.

Notons que si on ne veut pas affecter un coût unique à chaque bloc de base comme précisé dans le paragraphe précédent, il est aussi possible, par exemple, de considérer séparément les états abstraits de pipeline selon le (ou les) prédécesseurs. Dans ce cas, si un bloc de base BB possède deux prédécesseurs $pred1$ et $pred2$, il est possible de calculer le coût de BB quand le flux d'exécution provient de $pred1$ (respectivement $pred2$) par la formule $\hat{C}(\hat{p}_{pred1}^{sortie})$ (respectivement $\hat{C}(\hat{p}_{pred2}^{sortie})$), ceci permettra d'avoir deux temps, qui pourront par exemple être affectés aux arcs $(pred1, BB)$ et $(pred2, BB)$.

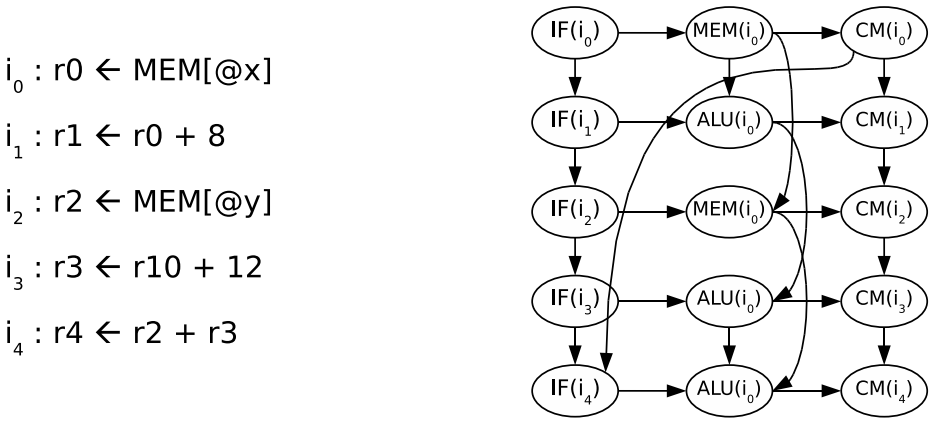
Ces résultats peuvent être facilement pris en compte avec la méthode IPET, y compris les cas où l'on souhaite prendre en compte le contexte sous forme d'une séquence de prédécesseurs, comme indiqué dans l'article [26].

2.5.3 La méthode des graphes d'exécution de C. Rochange et al.

Les graphes d'exécution (*exegraph*) sont une technique présentée dans [79], inspirée d'une méthode décrite dans l'article [52]. La méthode des graphes d'exécution permet de modéliser le contexte d'exécution de chaque bloc de base par un ensemble de paramètres. L'idée générale de la technique est basée sur le fait que, bien que le coût d'exécution du bloc de base puisse être exprimé en fonction de ces paramètres, il est possible de borner le pire coût indépendamment de la valeur des paramètres. Dans cette méthode, le coût d'exécution d'un bloc de base est défini comme le temps entre la fin de l'exécution de la première instruction du bloc de base, et la fin de l'exécution de la dernière instruction. Le coût d'exécution du premier bloc de base du programme est défini comme égal a son temps d'exécution.

Un graphe d'exécution est utilisé pour modéliser l'exécution d'un bloc de base. Ce graphe modélise notamment les dépendances entre les instructions du bloc de base, les contraintes liées au partage des ressources du processeur, et toutes les autres informations qui nous permettent de dériver le coût d'exécution du bloc de base de manière conservatrice.

A titre d'exemple, considérons le bloc de base B qui est représenté dans la figure 2.23 (a) par la liste de ses instructions en pseudo-code. Considérons qu'on est en présence d'un processeur scalaire simple avec un pipeline à trois étages : *fetch*, *execute*, *commit*, et deux unités fonctionnelles (une pour les calculs, et une pour l'accès à la mémoire). Nous supposerons également que les instructions sont exécutées dans l'ordre, et qu'il peut y en avoir au maximum quatre dans le pipeline à un instant donné. Le graphe d'exécution correspondant au bloc B est représenté dans la figure 2.23 (b). Chaque nœud



(a) Instructions (b) Graphe d'exécution

FIGURE 2.23: Exegraph : un exemple

du graphe correspond au traitement d'une instruction par un étage de pipeline ou une unité fonctionnelle. Les étages *fetch* et *commit* sont notés respectivement IF et CM, tandis que les deux unités fonctionnelles sont notées ALU et MEM. Les arcs dirigés du graphe correspondent à une relation de dépendance entre deux nœuds, c'est-à-dire qu'un arc ($n1$, $n2$) exprime le fait qu'une instruction, avant de pouvoir être traitée par l'étage ou l'unité fonctionnelle correspondant à $n2$, doit être traitée par $n1$. Par exemple, l'arc $(MEM(i_2), ALU(i_4))$ existe car l'addition (réalisée par l'ALU) de l'instruction i_4 nécessite le résultat de l'accès mémoire (réalisé par MEM) de l'instruction i_2.

Chaque nœud est associé à une latence lat_N, et l'instant auquel ce nœud est prêt est noté $prêt_N$ (l'instant auquel ce nœud est terminé est donc $lat_N + prêt_N$). Ceci permet de définir le coût d'un bloc de base B comme étant la différence entre l'instant de terminaison du dernier nœud du graphe d'exécution du bloc B (noté L_B) et du dernier nœud du bloc P prédécesseur de B (noté L_P), différence exprimée par $(prêt_{L_B} + lat_{L_B}) - (prêt_{L_P} + lat_{L_P})$.

Il est possible d'exprimer $prêt_N$ en fonction du contexte : pour ce faire, soit R l'ensemble des ressources dont le statut peut affecter le coût d'exécution d'un bloc de base (par exemple, la disponibilité d'une unité fonctionnelle ou d'une valeur de registre calculée par une instruction précédente). Le contexte d'exécution d'un bloc de base est spécifié par un vecteur $A = \{a^r | r \in R\}$, dans lequel chaque élément est un entier indiquant le moment où la ressource correspondante est libérée.

Chaque nœud est aussi associé à deux vecteurs (de même taille que A) pour définir les ressources dont il a besoin. Pour un nœud noté N, le vecteur $E_N = \{e_N^r \in \{0,1\} | r \in R\}$ contient un booléen associé à chaque ressource, pour indiquer si le nœud N dépend de l'état de la ressource. Le vecteur $D_N = \{d_N^r | r \in R\}$ contient un entier associé à chaque ressource, qui indique le temps (minimum) nécessaire entre la libération de la ressource et le moment où le nœud sera prêt à fonctionner. Pour un nœud N donné, on peut donc grâce à un contexte a, et aux deux vecteurs e_N et d_N, calculer l'instant auquel le nœud sera prêt, ce temps est donné par $prêt_N = max(e_N^r \times (d_N^R + a^r))$. Les deux vecteurs e_N et d_N sont initialisés pour chaque nœud en fonction des ressources qu'il demande (si un nœud N demande une ressource $r \in R$, alors e_N^r est mis à 1, et d_N^r à 0). Les vecteurs e_N

et d_N sont ensuite propagés des prédécesseurs vers les successeurs (en s'assurant de ne traiter un nœud que lorsque tous ses prédécesseurs ont été traités), grâce à un algorithme (détaillé dans [79]) que nous ne mentionnerons pas ici par souci de clarté.

Pour calculer le coût d'un bloc de base B, on pourrait à partir d'un contexte a, calculer $(prêt_{L_B} + lat_{L_B}) - (prêt_{L_P} + lat_{L_P})$. Toutefois, ceci n'est pas pratique car le contexte a n'est pas connu. Alors, l'article [79] montre qu'il est possible de borner le coût du bloc de base B, pour tout contexte a possible. Ceci permet donc de calculer un coût de B indépendant de a, mais en tenant compte du prédécesseur de B. Lorsqu'on calcule le coût de B, on peut donc prendre le pire temps parmi tous les prédécesseurs possibles, et donc affecter un temps unique à B, ce qui est quelque peu pessimiste. Mais on peut aussi affecter un coût à B qui est fonction du prédécesseur exécuté avant B (dans le cas où B possède plusieurs prédécesseurs), ce qui permet d'être moins pessimiste. Il est possible également de prendre en considération la séquence de prédécesseurs d'une longueur arbitraire n (plus précis mais plus coûteux au fur et à mesure que n augmente). Si plusieurs coûts de B sont calculés en fonction de la séquence de prédécesseurs, ceci devra être exprimé dans le système ILP de manière similaire à ce qui est fait dans la méthode des deltas, section . L'exemple présenté dans ce paragraphe correspond à un cas simple, les cas plus complexes (processeur superscalaire, exécution dans le désordre, etc.) font l'objet de traitements spécifiques décrits plus en détail dans [79].

2.6 Analyse de la prédiction de branchements

Lorsque le processeur rencontre une instruction de branchement conditionnel ou indirect, l'adresse destination n'est pas connue *a priori*. Ceci peut poser un problème au niveau du pipeline : l'instruction qui suit le branchement ne peut pas être chargée tant que l'adresse destination n'a pas été calculée. Pour éviter d'avoir à introduire un délai, la plupart des processeurs utilisent un mécanisme de prédiction de branchement. Lorsqu'une instruction de branchement conditionnel ou indirect est décodée, le prédicteur de branchement fournit une adresse destination prédite, et les instructions suivantes sont chargées à partir de cette adresse-là. Lorsque la véritable destination du branchement est calculée, si jamais la prédiction était mauvaise, il y a un vidage de pipeline et on reprend le chargement des instructions à partir de la bonne adresse, ce qui fait perdre du temps. Si la prédiction était bonne, il n'y a aucun délai.

Pour prédire l'adresse destination d'un branchement indirect, on utilise généralement un BTB (*Branch Target Buffer*). Ce BTB se comporte un peu comme un cache associatif par ensemble : chaque instruction de branchement, en fonction de son adresse, est projetée sur un ensemble du BTB, et si une des entrées de l'ensemble contient une adresse destination prédite, elle est récupérée grâce au tag (le concept de tag dans la BTB est équivalent à celui du tag de cache). Lorsque le processeur rencontre une instruction de branchement indirect, il recherche dans le BTB et utilise le résultat, le cas échéant, pour réaliser sa prédiction. Lorsque la destination réelle du branchement est connue, le BTB

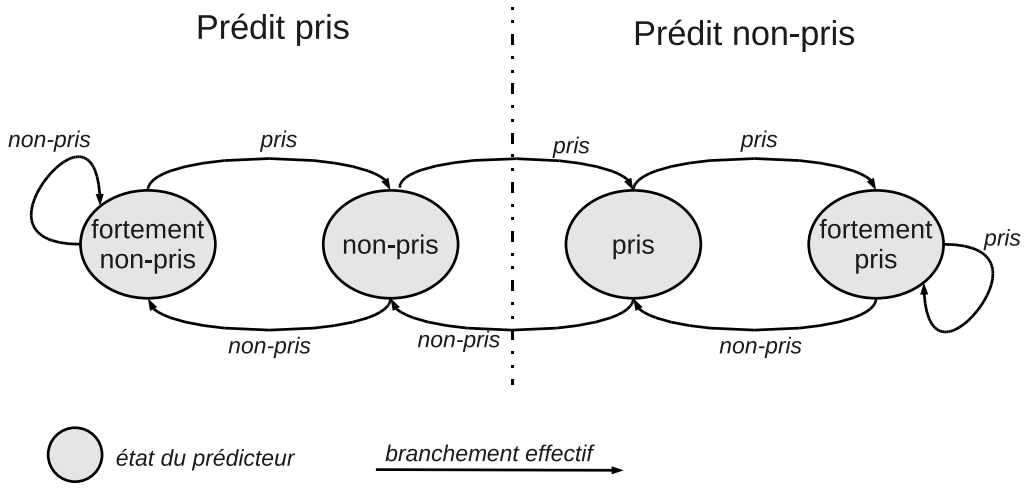


FIGURE 2.24: Prédicteur de branchement bimodal

est mis à jour.

Pour prédire si un branchement conditionnel est pris ou non pris, il y a plusieurs méthodes. La plus simple est la prédiction statique, qui se base uniquement sur la nature de l'instruction de branchement pour déterminer si elle sera prise ou non prise (par exemple : prédire "pris" les branchements vers l'arrière, et "non-pris" les branchements vers l'avant). Des méthodes plus évoluées utilisent l'historique des résultats des branchements. Par exemple, la prédiction bimodale utilise une BHT (Branch History Table) projetant chaque instruction de branchement sur une entrée de BHT contenant un état à quatre valeurs possibles : FP (Fortement Pris), P (Pris), NP (Non Pris) et FNP (Fortement Non Pris). Lorsque le processeur rencontre un branchement conditionnel et qu'une prédiction doit être réalisée, la BHT est examinée, et si le résultat est FP ou P alors le branchement est prédit pris, sinon il est prédit non-pris (il faut bien sûr sélectionner arbitrairement un état qui sera renvoyé en cas d'absence de l'instruction dans la BHT, cet état est le plus souvent P ou NP). Lorsque l'état (pris ou non-pris) d'un branchement conditionnel est réellement calculé, l'entrée de BHT est mise à jour en fonction de l'état précédent ainsi que du statut pris/non-pris du branchement actuel, tel qu'indiqué sur le schéma 2.24.

Cette technique possède l'inconvénient de mal prédire les branchements associés à la première et dernière itération de chaque boucle, mais donne globalement d'assez bons résultats et est assez simple à mettre en oeuvre.

2.6.1 Analyse de la prédiction de branchement par interprétation abstraite

Dans l'article [15], A. Colin et I. Puaut présentent une méthode de prise en compte de la prédiction de branchement pour le WCET. Cette approche prend en compte la

prédiction de branchement bimodale basée sur une BHT [3] associative par ensemble à A voies, et à politique de remplacement LRU.

Cette méthode est basée sur l'interprétation abstraite. Le domaine de cette interprétation abstraite est l'ABS (*Abstract BHT State*), chaque ABS représente un ensemble d'états possibles de la BHT lors de l'exécution du programme, et les ABS sont calculés par l'analyse en tout point du CFG.

Un ABS à l'entrée d'un bloc de base BB est noté $ABS_{BB}^{entrée}$. Pour un ABS donné abs, on notera $abs[l, a]$ l'ensemble des instructions de branchement qui peuvent être dans la BHT dans la ligne l et à l'âge a ($a \in [0..A[$).

La fonction *Join* de l'analyse de BHT est simplement l'union d'ensembles $ABS_3 = Join(ABS_1, ABS_2)$ tel que $\forall l, a : ABS_3[l, a] = ABS_1[l, a] \cup ABS_2[l, a]$. La fonction *Update* injecte dans l'ABS l'instruction de branchement rencontrée, le cas échéant, dans le bloc de base (un bloc de base ne peut avoir, au plus, qu'une instruction de branchement). Ainsi, l'*Update* est défini par $ABS_{out} = Update(ABS_{in}, instr)$ tel que $\forall l \neq ligne_{instr} \forall a$, $ABS_{out}[l, a] = ABS_{in}[l, a]$. Dans le cas $l = ligne_{instr}$ on a :

- $ABS_{out}[ligne_{instr}, 0] = \{instr\}$
- $ABS_{out}[ligne_{instr}, a] = ABS_{in}[ligne_{instr}, a - 1]$ pour tout $a \in [1..A[$

Une fois les ABS calculés en tout point du CFG, il est nécessaire de connaître le niveau d'imbrication de boucle de chaque instruction de branchement, noté $lnlevel(instr)$ (*Loop Nesting Level).* Le *lnlevel* d'une instruction fait référence à la boucle la plus interne contenant l'instruction. L'ensemble des boucles forme un ensemble partiellement ordonné par la relation "L_1 est incluse dans L_2" notée $L_1 \preceq L_2$.

Une fois ceci fait, à partir du *lnlevel* et de l'ABS_{in} de chaque instruction de branchement, il est possible de les classer en quatre catégories :

- *Always D-Predicted* signifie que l'instruction sera toujours prédite grâce à la prédiction par défaut (qui est le plus souvent prédit pris). Cette catégorie est affectée à une instruction *instr1* lorsque le résultat de l'interprétation abstraite implique que l'instruction ne sera jamais dans la BHT lors de son exécution, ou bien lorsque l'instruction est en conflit avec une autre instruction $instr2$, et que le *lnlevel* de cette dernière implique qu'elle est forcément exécutée entre deux exécutions de $intsr1$.
- *First D-Predicted* signifie que l'instruction sera toujours prédite grâce à la prédiction par défaut lors de sa première exécution, mais sera prédite grâce à l'historique lors des exécutions suivantes. Cette catégorie est affectée à une instruction $instr1$ lorsque le résultat de l'interprétation abstraite montre que cette instruction est seule dans $ABS_{out}[ligne_{instr}, *]$. En effet, cela signifie qu'une fois que cette instruction est dans la BHT, aucune autre instruction ne peut l'en déloger, et que donc cette instruction ne peut être D-Predicted qu'une fois. Une version plus faible de cette propriété existe lorsqu'une instruction $instr1$ est D-Predicted une fois par itération d'une boucle extérieure donnée, par exemple si cette boucle contient une instruction en

3. l'article parle en fait de BTB, mais apparemment la modélisation qui est faite de ce composant laisse penser qu'il s'agit d'une table hybride BHT/BTB, associant à chaque instruction à la fois l'état bimodal et l'adresse destination en cas de branchement indirect.

conflit avec $instr1$. Dans ce cas, une nouvelle valeur $Clevel_{instr1}$ est calculée, faisant référence à la boucle en question.

- *First Unknown* signifie que l'instruction sera prédite soit par la prédiction par défaut soit par l'historique lors de sa première exécution, mais sera prédite grâce à l'historique lors des exécutions suivantes. Cette catégorie est forcément associé à un $Clevel$, et apparaît lorsque l'instruction $instr1$ considérée est en conflit avec une autre instruction dans une boucle extérieure, uniquement si on n'est pas sûr de passer par $instr2$ à chaque itération de la boucle extérieure.
- *Always Unknown* signifie qu'on ne sait pas si l'instruction sera prédite grâce à la prédiction par défaut ou grâce à l'historique. Cette catégorie est le cas par défaut, lorsqu'aucun des autres cas ne s'applique pour l'instruction considérée.

Ensuite, ces catégories peuvent être utilisées pour prendre en compte dans le WCET les pénalités dues aux mauvaises prédictions de branchement.

2.7 Benchmarks de WCET

Bien que certains articles traitent du calcul de WCET sur des applications réelles [13, 47], il existe également de petit programmes qui servent couramment de jeux de tests (benchmarks) lors des expérimentations dans le cadre du calcul de WCET. Ces tests sont généralement assez simples, et présentent les bonnes caractéristiques qui permettent de calculer le WCET facilement. Il existe plusieurs jeux de tests, certains sont librement accessibles, voici quelques exemples :

- Les benchmarks SNU-RT [82] (Seoul National University - Real Time) sont un ensemble de petits programmes de test spécifiquement créés pour faire du calcul de WCET. Ils sont principalement issus d'algorithmes de traitement de signal numérique, et ont été modifiés pour rendre le calcul de WCET possible. Notamment, il n'y a pas de sauts inconditionnels ou de sorties depuis l'intérieur du corps des boucles, pas de *switch* ni de *do/while* ni d'appel de fonction de librairie.
- Les benchmarks Malärdalen [62], publiés par le MRTRC (Malärdalen Real-Time Research Center) de l'Université de Malärdalen (Suède), sont du même type que les benchmarks SNU-RT, dont il sont un sur-ensemble. Mais, par rapport aux benchmarks SNU-RT, les benchmarks Malärdalen ont été enrichis de quelques programmes supplémentaires, un peu plus grands et plus intéressants pour tester du calcul de WCET.
- Les benchmarks MiBench, proposés par l'Université de Michigan, consistent en un ensemble de 35 programmes embarqués temps-réel, ces programmes sont conçus pour les tests tout en essayant d'être représentatifs de programmes embarqués temps-réel utilisés dans diverses industries (automobile, par exemple). Dans l'article [39], MiBench est comparé aux benchmarks SPEC2000 (Standard Performance Evaluation Corporation), un jeu de benchmarks fréquemment utilisés.
- Le benchmark PapaBench est issu d'une application réelle, un logiciel embarqué

temps-réel utilisé pour contrôler un drone (projet Paparazzi). Ce benchmark est conçu pour l'expérimentation dans le calcul de WCET, mais peut aussi être utile pour d'autres domaines (comme l'analyse d'ordonnancement). Dans le papier [68], PapaBench est présenté, et comparé à des séries de benchmarks existants.

Dans le cadre des expérimentations de cette thèse, nous utiliserons principalement les benchmarks Malärdalen car ils sont assez diversifiés (incluant les tests SNU-RT), certains d'entre eux sont assez importants pour être intéressants du point de vue du calcul de WCET, sans toutefois être trop volumineux ce qui permet de garder un temps de calcul réduit.

Chapitre 3

Analyse de la hiérarchie mémoire

Ce chapitre présente nos travaux sur la hiérarchie mémoire, en particulier l'amélioration de l'analyse de *persistence* sur le cache d'instructions, ainsi que le développement d'une analyse du *row buffer*.

3.1 Le First miss

Dans cette partie, nous présentons les travaux que nous avons réalisés sur l'amélioration du traitement des l-blocs *First Miss* dans le cadre de l'analyse du cache d'instructions.

3.1.1 Limitations du déroulage de boucles

La technique du déroulage de boucles possède deux inconvénients principaux. Le premier inconvénient est illustré dans la figure 3.1. Dans cette figure, par souci de simplicité, chaque bloc de base est contenu dans un seul bloc de cache (c'est-à-dire que les blocs de base sont aussi des l-blocs). De plus, on n'a numéroté que les l-blocs projetés sur l'ensemble de cache en cours d'analyse, car pour la plupart des analyses de cache d'instructions, chaque ensemble peut être traité indépendamment (nous adoptons cette dernière convention pour tous les autres schémas similaires sur l'analyse de cache). Par convention, nous indiquons les têtes de boucles par Tn (où n est le numéro de boucle dans le cas où le schéma comporte plusieurs boucles), et nous signalons les versions déroulées par Tn'. Sur la figure, le l-bloc 0 ne peut pas être catégorisé AH dans le contexte « *autres itérations* » à cause du branchement conditionnel : bien qu'on sache que ce l-bloc ne peut pas causer plus d'un seul *miss,* on n'est pas sûr que la première exécution de ce l-bloc se produira lors de la première itération de la boucle. Par conséquent, l'analyse de cache avec déroulage de boucle catégorisera ce l-bloc NC, ce qui provoque du pessimisme au niveau du WCET.

Le deuxième inconvénient concerne la taille du système ILP et le coût des analyses statiques. En effet, le déroulage de boucle duplique les blocs (encore plus quand il y a un grand nombre de boucles imbriquées), ce qui augmente énormément la taille du système ILP. La résolution du système ILP est très coûteuse (c'est l'étape la plus longue dans le calcul de WCET) et son temps augmente de manière non linéaire par rapport au nombre

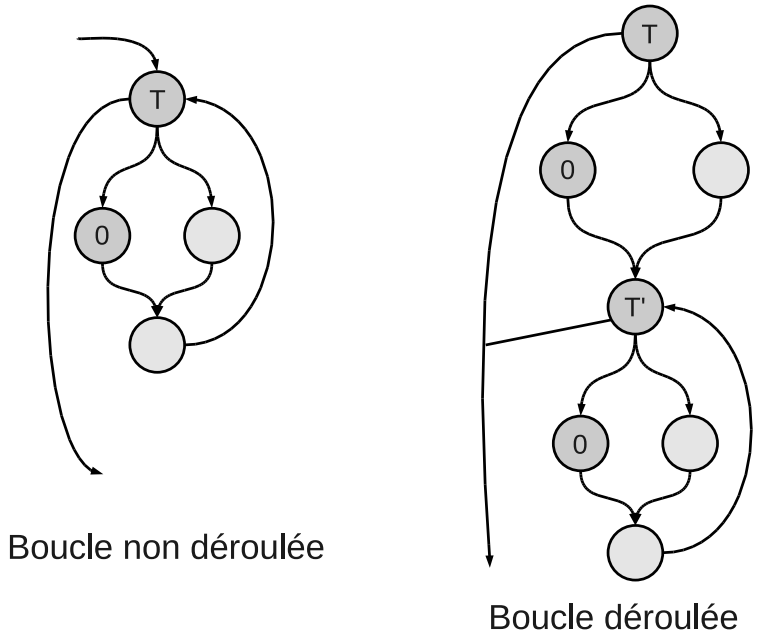


FIGURE 3.1: Inconvénient du déroulage de boucles

de variables/contraintes du système. C'est pourquoi, même si l'analyse de cache avec déroulage de boucles est en soi assez rapide, l'augmentation de la taille du CFG et du système ILP qu'elle provoque augmente beaucoup le temps d'analyse global.

3.1.2 Limitations de l'analyse de *persistence*

L'analyse de *persistence* permet de gérer les blocs de type « *First Miss* » rapidement, et sans avoir recours au déroulage de boucles. Toutefois cette approche souffre d'un manque de précision en présence de boucles imbriquées, comme on peut s'en rendre compte dans l'exemple de la figure 3.2. L'exemple montre deux boucles imbriquées, ayant pour en-têtes H_1 (boucle externe) et H_2 (boucle interne). La première itération de la boucle interne est déroulée, car on veut montrer que le déroulement de boucles ne résout pas le problème qu'on exposera dans ce paragraphe. On voit sur le schéma que le bloc de cache 0 est *persistent* par rapport à la boucle interne : une fois qu'il est chargé, il ne peut pas être remplacé pendant tout le déroulement de cette boucle. Toutefois, ce bloc est en conflit avec les deux blocs foncés de la boucle externe, c'est pourquoi l'analyse de *persistence* n'affecte pas la catégorie PS pour le bloc 0. De plus, ce bloc est dans un chemin conditionnel, donc pour les raisons exposées lors de la section précédente, il ne sera pas catégorisé AH dans le contexte "*autres itérations*" de la boucle interne.

Dans cet exemple, on voit que même l'analyse de *persistence* et le déroulage de boucles combinés ne parviennent pas à tirer parti du fait que le bloc 0 ne peut pas être remplacé dans la boucle interne. C'est pourquoi une nouvelle technique d'analyse serait intéressante afin d'éliminer ces inconvénients.

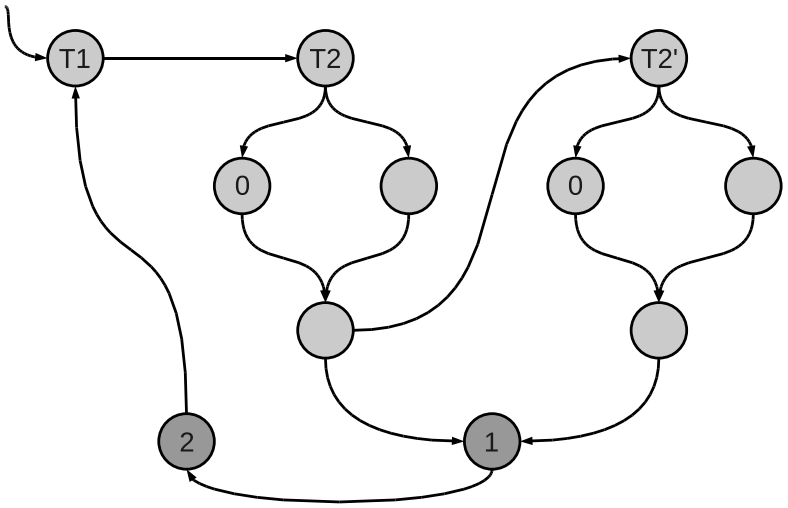


FIGURE 3.2: Inconvénient d'analyse de *persistence*

3.1.3 Notre approche

En raison du problème de l'explosion de contraintes ILP due au déroulage de boucles, et du problème de précision de l'analyse de *persistence* (et ce, même lorsqu'elle est utilisée avec le déroulage de boucles), nous avons développé une version améliorée de l'analyse de persistence, qui est décrite dans [5]. Une analyse de *persistence* suffisamment précise permettrait de se passer du déroulage de boucles, et donc fournirait une solution aux deux problèmes.

3.1.3.1 *Persistence* interne

Pour résoudre le problème présenté dans la figure 3.2, il faut une version de l'analyse de *persistence* qui peut traiter les boucles internes, en cas d'imbrication. Nous avons défini une nouvelle version de la *persistence* nommée « *Persistence* interne » pour la différencier de l'ancienne analyse de *persistence* (que nous nommerons « *Persistence* externe » à partir de maintenant). Un l-bloc est considéré « *Persistent* interne » si, lorsqu'il est chargé, il ne peut pas être remplacé durant l'exécution de la boucle la plus interne contenant ce l-bloc. Pour pouvoir calculer la propriété de *persistence* interne, il faut modifier l'analyse pour que lorsqu'on entre dans une boucle, on mette l'ACS à $\perp$ (l'état vide). Par conséquent, l'analyse de la boucle interne prend en compte seulement les remplacements de blocs qui ont lieu dans cette boucle interne. Toutefois, si on emploie cette méthode, lorsqu'on quitte la boucle un problème se pose (comme précisé dans la figure 3.3) : on ne peut pas correctement mettre à jour les successeurs de l'arc de sortie car on ne sait pas ce qui est arrivé à l'état du cache pendant l'analyse de la boucle.

Pour régler ce problème, il faudrait analyser en parallèle l'état général et $\perp$ pendant le traitement de la boucle (on pourra ensuite éliminer les résultats de l'analyse à partir de $\perp$ lorsqu'on quittera la boucle) et ainsi avoir les ACS corrects en sortie. Si on généralise ce principe à une imbrication de boucle à n niveaux, on doit traiter jusqu'à n ACS en parallèle. De plus, bien que cette approche résolve le problème présenté dans la figure 3.3,

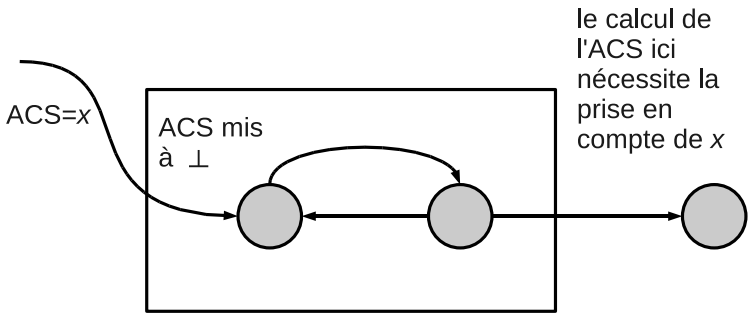


FIGURE 3.3: *Persistence* interne

la propriété de *persistence* interne est plus faible que celle de *persistence* externe. En effet, lorsqu'un l-bloc est *persistent* externe, il causera au pire un seul *miss* pour tout le programme. Or si un l-bloc est *persistent* interne, il causera potentiellement un *miss* chaque fois qu'on rentre dans la boucle interne. C'est pourquoi cette approche peut ajouter du pessimisme dans le cas où un l-bloc est réellement *persistent* externe. Toutefois, on peut noter que dans cette analyse, lorsqu'on traite une boucle L, on traite aussi les informations qui concernent les boucles qui contiennent L. Donc on a les informations qui nous permettent de dire si un l-bloc dans L est *persistent* interne ou externe. En fait, on a même les moyens de dire exactement, pour chaque l-bloc et pour chaque boucle le contenant, si le l-bloc pourra ou non être remplacé dans la boucle. L'analyse de *persistence* paramétrique présentée dans la prochaine section permet de mettre à profit cet avantage.

3.1.3.2 *Persistence* paramétrique

Dans cette approche, la catégorie *Persistent* est paramétrique, c'est-à-dire qu'elle est associée à une boucle. Nous catégoriserons un l-bloc $Persistent(L_1)$ si une fois qu'il est chargé, il ne peut pas être remplacé durant l'exécution de la boucle L_1 (et donc durant l'exécution des boucles internes de L_1). Un l-bloc $Persistent(L_1)$ peut provoquer au pire un *miss* chaque fois qu'on rentre dans L_1. Cette approche est plus coûteuse en temps de calcul (ce qui est largement compensé par l'élimination du déroulage de boucles), mais ceci nous permet d'avoir le niveau exact de conflit du bloc, pour une précision largement améliorée. En fait, les persistences internes et externes ne sont que des cas particuliers de *persistence* paramétrique : si un l-bloc *lb* est *persistent* interne (resp. persistent externe), alors il est $Persistent(L_i)$, où L_i est la boucle la plus externe (resp. la plus interne) contenant *lb*. Ceci permet d'être sûr que la *persistence* paramétrique est au moins aussi bonne que les persistences internes et externes (mais la plupart du temps elle est meilleure).

Pour calculer la *persistence* paramétrique, on utilise un ACS qui est en fait une pile pour laquelle chaque élément est un ACS de *persistence* classique. Pour chaque l-bloc *lb*, l'ACS de *persistence* paramétrique associe un élément de la pile à chaque boucle L_i qui contient *lb*. Nous noterons cet élément $ACS^B_{pers}[Li]$, il représente l'information de *persistence* avant le l-bloc *lb*, calculée en prenant en compte uniquement les remplacements

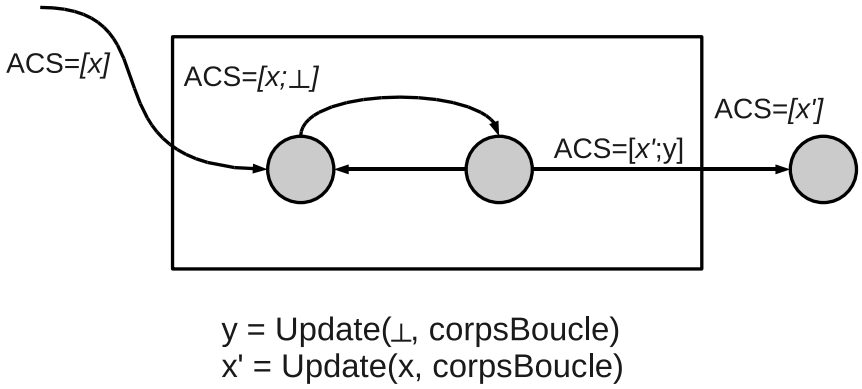


FIGURE 3.4: *Persistence* paramétrique

de blocs de cache qui ont lieu dans la boucle L_i. Les fonctions Join et Update sont dérivées de la *persistence* classique, mais modifiées pour travailler sur la pile, en réalisant les Join et Update sur les éléments correspondants de la pile, comme montré dans les formules de la figure 3.4 (les fonctions $Update_{ext}$ et $Join_{ext}$ représentent respectivement les fonctions Update et Join de la *persistence* externe classique).

La pile est initialement vide au début du programme. Lorsqu'on entre dans une boucle, un nouvel élément vide est empilé. Lorsqu'on quitte une boucle, on dépile et élimine le plus haut élément de la pile. Pour déterminer l'existence de l-blocs de catégorie *persistent* à partir des piles d'ACS calculées par l'analyse, définissons d'abord la fonction $IsPers()$, tel que $IsPers(lb, L_i) = \exists a/block_{lb} \in ACS^{lb}_{pers}[L_i](l, a) \wedge a \neq l_\top$. Cette fonction teste si le l-bloc lb est $Persistent(L_i)$, et renvoie une valeur booléenne. A partir de cette fonction, le calcul de la catégorie est définie par la règle suivante (calcule la boucle la plus externe à partir de laquelle lb est *persistent*) :

$$cat_{lb} = \begin{cases} Persistent(L_i) & si\ IsPers(lb,\ L_i) \wedge \\ & \forall L_j,\ L_i \subset L_j \rightarrow \neg IsPers(lb, L_j) \\ Not\,Classified & sinon \end{cases}$$

3.1.3.3 Autres analyses permettant d'éviter le loop unrolling

Un des buts importants de la *persistence* paramétrique est d'éviter d'avoir à réaliser du déroulage de boucles. Toutefois, certains effets de cache sont pris en compte lorsqu'on fait du déroulage de boucles, mais ne le sont pas lorsqu'on utilise la *persistence* paramétrique. C'est pourquoi nous avons dû concevoir deux raffinements de l'analyse pour gérer ces cas.

Le premier raffinement, l'analyse des l-blocs liés, a été réalisé à cause d'une situation illustrée dans la figure 3.5(a). Dans ce schéma, les l-blocs 1a et 1b sont tous deux dans le bloc de cache 1. Comme prévu, grâce au déroulage de boucles les l-blocs 1a et 1b sont catégorisés NC pour la première itération, et AH pour les autres itérations. Si le déroulage de boucles n'est pas utilisé mais qu'on utilise la *persistence* paramétrique, les deux l-blocs sont catégorisés PS, ce qui était attendu.

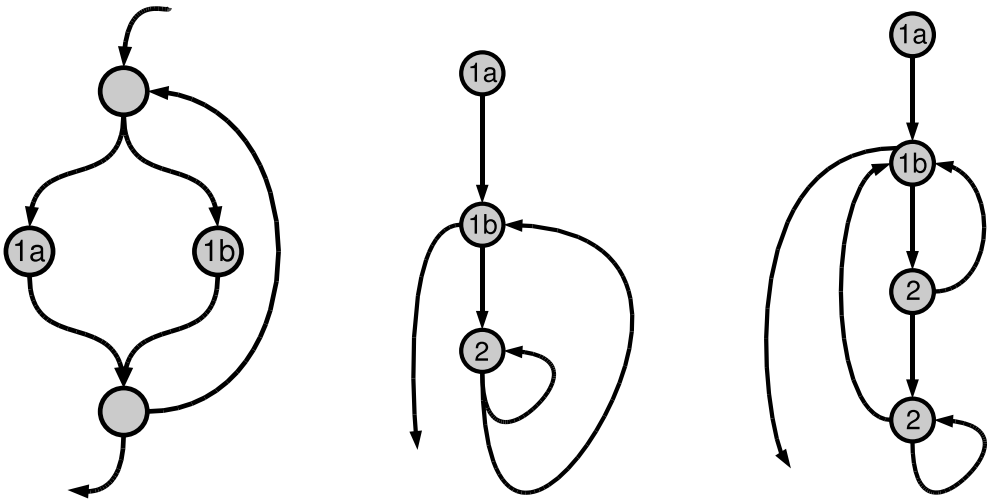
Jusqu'ici, la raison pour laquelle la deuxième approche introduit du pessimisme n'est pas évidente. La différence est due au fait que dans le cas du déroulage de boucles, on sait qu'on aura au pire un seul *miss* chaque fois qu'on entre dans la boucle (puisque les l-blocs 1a et 1b appartiennent au même bloc de cache) ; toutefois dans le cas de la *persistence* paramétrique on prédira au pire deux *miss* (un pour 1a, et un pour 1b), bien que cela soit impossible car 1a et 1b partagent le même bloc de cache.

Pour résoudre ce problème, une nouvelle analyse est introduite pour identifier les ensembles de l-blocs de catégorie *persistent* par rapport à la même boucle et partageant le même bloc de cache. C'est assez simple : une fois que l'analyse de *persistence* paramétrique est terminée, on parcourt tout les blocs *persistents*, puis on regroupe les blocs qui sont *persistents* par rapport à la même boucle et qui sont dans le même bloc de cache. Ceci nous permet de générer des contraintes ILP pour prendre en compte l'effet décrit dans la figure 3.5(a). Pour cet exemple spécifique, une analyse de *persistence* paramétrique permettrait de générer les contraintes $x^{1a}_{miss} \leq 1$ et $x^{1b}_{miss} \leq 1$ (signifiant que le l-bloc 1a ne peut pas provoquer plus d'un *miss,* et qu'il en va de même pour le l-bloc 1b, mais les deux peuvent causer conjointement deux *miss*). Par contre, si on exécute l'analyse des blocs liés, on peut générer une contrainte unique $x^{1a}_{miss} + x^{1b}_{miss} \leq 1$, signifiant que les l-blocs 1a et 1b *ensemble* ne peuvent pas générer plus d'un seul *miss*.

Le deuxième raffinement a été conçu en réponse à un problème exposé dans les figures 3.5(b) et 3.5(c). Les deux figures montrent le même CFG, mais la version de droite a été soumise au déroulage de boucles. Les l-blocs 1a et 1b sont dans le même bloc de cache. Cet exemple ne concerne pas directement la *persistence* car il est illustré à l'aide de l'analyse *Must*, mais il met en lumière un pessimisme qui apparaît dès qu'on ne fait pas de déroulement de boucles. Dans cet exemple, on voit aisément que le l-bloc 1b sera catégorisé AH. En effet, il y a deux chemins qui aboutissent à 1b : le chemin venant de 1a, ainsi que le chemin passant par 2 et par l'arc de retour de la boucle. Si l'on emprunte le premier chemin, le l-bloc 1b sera un *hit* parce que le bloc de cache 1 est chargé lorsqu'on passe par *1a*. Pour le second chemin, puisque un seul autre bloc de cache est référencé dans la boucle (bloc de cache numéro 2) et que le cache est associatif par ensemble à 2 voies, le bloc de cache 1 n'est pas remplacé.

Le problème est le suivant : lorsque l'analyse *Must* est réalisée sans déroulage de boucles, le l-bloc 1b n'est pas catégorisé AH. Quand l'analyse passe pour la première fois par le l-bloc 1b, elle trouve $ACS^{1b\to 2}_{must}(l) = [1, \emptyset]$. L'*update* pour le l-bloc 2 produit $ACS^{2\to 2} = [2, 1]$. Puis, le Join entre les deux ACS avant le l-bloc 2 produit $ACS^{2}_{must}(l) = [\emptyset, 1]$. Ensuite, l'Update du l-bloc 2 produit le résultat $ACS^{2\to 1} = ACS^{2\to 2} = [2, 0]$, dans lequel le l-bloc 1 est absent, ce qui empêche 1b d'être catégorisé en tant que AH.

On voit facilement que le problème vient de l'union des deux arcs entrants du l-bloc 2. Puisque le résultat est $ACS^{2}_{must}(l) = Join([1, \emptyset], [2, 1]) = [\emptyset, 1]$, on en conclut (à tort) qu'il est possible que le bloc de cache 1 soit remplacé par le passage par le l-bloc 2. Toutefois, si



(a) Problème 1

(b) Problème 2 (non déroulé)

(c) Problème 2 (déroulé)

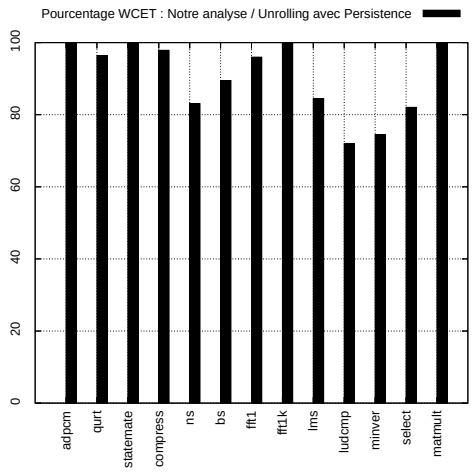
FIGURE 3.5: Raffinements d'analyse de *persistence*

on examine les ACS avant le Join, on se rend compte que ce n'est pas possible : soit le bloc de cache 1 est présent à l'âge 0 et son âge est simplement incrémenté, soit il est à l'âge 1 mais son âge ne change pas puisque le bloc référencé (2) est déjà présent dans le cache. Cette information est perdue lors du Join, sauf dans le cas du déroulage de boucles : dans ce cas, l'arc entrant et l'arc de retour de la boucle arrivent sur des instances distinctes du l-bloc 2 et le problème est évité.

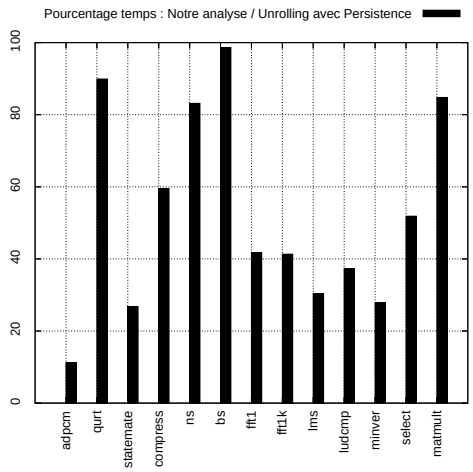
Il serait intéressant de pouvoir bénéficier des avantages du déroulage de boucles pendant le calcul des ACS, tout en évitant le problème de l'explosion du nombre de contraintes ILP dû à cette approche. C'est pourquoi nous avons choisi de faire du déroulage de boucles avant le calcul des ACS, ce qui nous permet d'éviter le problème du Join décrit plus haut. Une fois que les ACS ont été calculées, on repasse sur le CFG normal (non déroulé) : chaque ACS du CFG non déroulé est calculée en faisant l'union des ACS correspondantes dans le CFG déroulé. Ceci permet d'avoir les avantages du déroulement de boucles (élimination du pessimisme sur le calcul des ACS) sans en avoir les inconvénients (augmentation du nombre de contraintes ILP) puisque le CFG utilisé pour la génération des contraintes structurelles est non déroulé.

3.1.4 Résultats

Nous comparons ici la performance et la précision de notre analyse (*persistence* paramétrique, avec pseudo-déroulage de boucles et analyse des blocs liés) avec l'analyse pré-existante de C. Ferdinand et al. (*persistence* avec déroulage de boucles). Les figure 3.6 montrent les divers résultats (en terme de temps et de précision de WCET) de la comparaison. Les mesures ont été réalisées grâce à OTAWA, sur un sous-ensemble des benchmarks Malärdalen. L'architecture cible était un processeur pipeliné simple (les effets de recouvrement de blocs étaient pris en charge par la méthode des deltas de J. Engblom et al.), accompagné d'un cache d'instructions de 1Ko associatif par ensemble à 4 voies.



(a) Mesures du pessimisme



(b) Mesures du temps d'analyse

FIGURE 3.6: Expérimentations sur la *persistence*

La figure 3.6(a) mesure la précision du WCET de chaque méthode. Pour chaque benchmark de Malärdalen que nous avons employé, le diagramme montre le pourcentage exprimant le ratio du WCET calculé avec l'analyse de *persistence* paramétrique (améliorée grâce à l'analyse des blocs liés et au pseudo-déroulage de boucles), par rapport au WCET trouvé par l'analyse de C. Ferdinand et al. (c'est-à-dire que chaque barre représente $\frac{WCET_{notreAnalyse}}{WCET_{Ferdinand}} \times 100$). Le diagramme 3.6(b) compare de manière similaire le temps pris par les deux analyses pour chaque méthode. Dans chaque cas, le temps total de calcul du WCET est comptabilisé, y compris le temps passé pour la résolution du système ILP.

Concernant la précision, on voit que notre analyse est plus précise que l'analyse de *persistence* traditionnelle avec déroulage de boucles. Grâce à ceci, nous n'avons plus besoin du déroulage de boucles. En moyenne, le WCET est réduit de 6.68%. Concernant le temps de calcul, notre analyse de *persistence* prend plus de temps que l'analyse de *persistence* classique, mais le temps perdu est largement récupéré par le gain de temps lors de la résolution du système ILP. En moyenne, sur la durée totale du calcul du WCET, notre analyse est 2.7 fois plus rapide que l'analyse de *persistence* classique avec déroulage de boucles.

3.2 Travaux sur le *row buffer*

Dans cette partie, nous expliquons comment nous avons pris en compte le *row buffer*, qui est présent dans les mémoires DRAM (Dynamic Random Access Memory).

3.2.1 Définition du problème

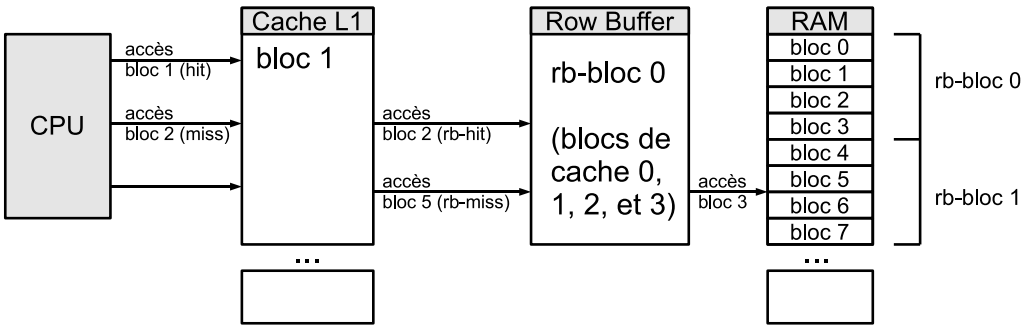


FIGURE 3.7: Le *row buffer*

Le *row buffer* que nous avons étudié (dans l'article [9]) se situe entre le cache d'instruction de premier niveau et la mémoire centrale, comme présenté dans le schéma 3.7. Il se comporte essentiellement comme un cache à application directe qui ne posséderait qu'une ligne. Nous supposons que la taille de bloc du *row buffer* est un multiple de la taille de bloc de cache, ceci étant une hypothèse réaliste.

Lorsqu'une instruction est chargée par le processeur, d'abord le bloc de cache correspondant est cherché dans le cache d'instructions. S'il est présent, alors un *hit* de cache se produit, et le processeur obtient l'instruction. Sinon, la requête est envoyée à la mémoire qui est dans notre cas une DRAM. Dans ce cas, si le bloc de *row buffer* correspondant à l'instruction est présent dans le *row buffer* alors nous avons un *hit* de row buffer, sinon la ligne de DRAM doit être chargée depuis la table des cellules mémoires et c'est un *miss*. Puisque l'accès au *row buffer* dépend du résultat de l'accès au cache (*hit* ou *miss*), l'analyse du *row buffer* doit prendre en compte l'analyse du cache d'instructions.

Pour une instruction notée *instr*, nous noterons $lbloc_{instr}$ son *l-bloc*, et nous noterons $rbbloc_{instr}$ son bloc de *row buffer*, appartenant à RB (clarification : $rbbloc_{instr}$ est l'équivalent du $bloc_{instr}$ du cache pour le *row buffer*, et non l'équivalent de $lbloc_{instr}$)

3.2.2 Notre approche

Notre approche, présentée dans le papier [9], est basée sur l'interprétation abstraite. Soit RB l'ensemble des blocs de *row buffer* qui constituent le programme à analyser. Tout d'abord, définissons le domaine concret : à chaque instant le *row buffer* contient exactement un bloc. C'est pourquoi le domaine concret D est en fait équivalent à RB.

La fonction *Update* concrète, de type $INSTR \times D \rightarrow D$, a besoin de connaître le résultat de l'accès de cache du bloc contenant l'instruction. En cas de *hit*, $Update(instr, d) = d$. Par contre, en cas de *miss*, on a $Update(instr, d) = rbbloc_{instr}$.

L'approche utilisée pour le *row buffer* est divisée en deux analyses distinctes : le *must*, et le *may*. Le domaine abstrait $D\prime$ est 2^{RB}, c'est-à-dire l'ensemble des blocs qui peuvent être (analyse *may*) ou bien qui doivent être (analyse *must*) dans le *row buffer*.

La fonction *Update* abstraite est une extension de la fonction *Update* concrète au domaine D'. Elle est donc de type $INSTR \times D\prime \rightarrow D\prime$, et a besoin de connaître le résultat de la catégorisation du *l-bloc* contenant l'instruction. Si la catégorie de $lbloc_{instr}$

est *Always Hit* alors le *row buffer* n'est pas utilisé, et on a donc $Update(instr, d\prime) = d\prime$. Si la catégorie de $lbloc_{instsr}$ est *Always Miss* alors on a $Update(instr, d\prime) = \{rbbloc_{instr}\}$. S'il s'agit d'une autre catégorie (*Not Classified*, ou *Persistent/First Miss* par exemple), alors on ne sait pas si l'accès au cache sera un *hit* ou un *miss,* donc on fait l'union des deux cas : $Update(instr, d\prime) = d\prime \cup \{rbblock_{instr}\}$.

Comme d'habitude, la fonction *Update* abstraite peut être étendue à tout un bloc de base par composition, et la fonction *Join* est l'union ensembliste pour le *may*, et l'intersection pour le *must*. L'interprétation abstraite calcule donc des états abstraits de *row buffer* (*ARS* - Abstract Row buffer State) en tout point du CFG. Une fois ceci fait, on peut classer chaque l-bloc selon quatre catégories de type de comportement vis à vis du *row buffer* :

- La catégorie *Row buffer Never Used* (RNU) est utilisée lorsque ce l-bloc ne cause pas d'utilisation du *row buffer*. Cette catégorie est utilisée uniquement lorsque la catégorie de cache du l-bloc est *Always Hit*. Par conséquent, dans toutes les autres catégories ci-dessous, on fait l'hypothèse que la catégorie de cache n'est pas *Always Hit*.
- La catégorie *Row buffer Always Hit (RAH)* est utilisée lorsque tout accès au *row buffer* concernant ce l-bloc sera un *hit*. Cette catégorie est utilisée lorsque $rbblock_{instr} \in ARS^{instr}_{must}$.
- La catégorie *Row buffer Always Miss* (RAM) est utilisée lorsque tout accès au *row buffer* concernant ce l-bloc sera un *miss*. Cette catégorie est utilisée lorsque $rbblock_{instr} \notin ARS^{instr}_{may}$.
- La catégorie *Row buffer Not Classified* (RNC) est utilisée lorsqu'on ne peut pas déduire d'informations sur le comportement du *row buffer* (c'est-à-dire lorsque les conditions nécessaires pour les trois catégories ci-dessus ne s'appliquent pas).

Pour prendre ceci en compte au niveau d'IPET, il faut considérer qu'il y a trois possibilités pour chaque l-bloc : un *hit*, un cache *miss (cmiss)* ou bien un *row buffer* miss (*rmiss*), contre deux possibilités (cache *hit* ou cache *miss*) dans le cas d'un système sans *row buffer*. Reprenons le même principe de prise en compte du cache avec IPET présenté dans la section 2.4.3.3. Chaque variable de type x^i_{miss} doit être supprimée, et remplacée par $x^i_{cmiss} + x^i_{rmiss}$ (respectivement, nombre de *miss* de cache avec *hit* de *row buffer* et nombre de *miss* de cache avec *miss* de *row buffer*). Partout dans le système ILP tel que construit après l'analyse du cache, on fait ce remplacement. Ceci entraîne que pour chaque l-bloc on doit toujours avoir $x^i = x^i_{hit} + x^i_{cmiss} + x^i_{rmiss}$. Dans la fonction objectif, il faut assigner un coefficient différent aux variables x^i_{cmiss} et x^i_{rmiss}, puisque le temps nécessaire en cas de *row buffer miss* est plus important.

Ensuite, pour éviter que le solveur ILP ne maximise systématiquement les variables x^i_{rmiss} et ne laisse toute les variables x^i_{cmiss} à 0, il faut contraindre x^i_{cmiss} et x^i_{rmiss} en fonction de la catégorie *row buffer* du l-bloc i :

- La catégorie *Row buffer Never Used* implique que $x^i_{cmiss} = x^i_{rmiss} = 0$. Cette contrainte ILP est déjà générée par le traitement de la catégorie de cache *Always Hit*.

En effet, la contrainte $x^i_{miss} = 0$, une fois x^i_{miss} remplacé, devient $x^i_{cmiss} = x^i_{rmiss} = 0$. C'est pourquoi aucune contrainte supplémentaire n'est nécessaire pour traiter la catégorie *Row buffer Never Used.*

- La catégorie *Row buffer Always Hit* provoque l'ajout de la contrainte $x^i_{rmiss} = 0$, indiquant que tous les *miss* de cache sont des *hit* de *row buffer*.
- La catégorie *Row buffer Always Miss* provoque l'ajout de la contrainte $x^i_{cmiss} = 0$, indiquant que tous les *miss* de cache sont aussi des *miss* au niveau du *row buffer*.
- La catégorie *Row buffer Not Classified* ne provoque pas l'ajout de contraintes.

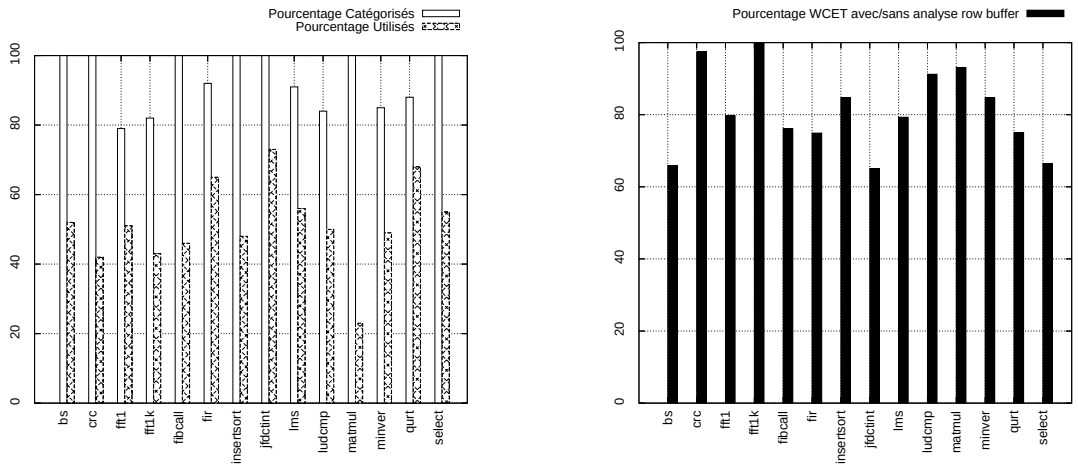
Cette analyse suffit pour avoir une estimation de l'impact du *row buffer* dans le WCET. Il n'est pas très intéressant de faire une analyse de *persistence* pour le *row buffer*. En effet, pour qu'un bloc soit *persistent* au niveau du *row buffer*, il faudrait qu'il soit dans une boucle qui tient entièrement dans le *row buffer*. Comme le cache d'instruction est en général plus grand que le *row buffer* (et ce même si l'on divise la taille du cache par le nombre de voies de celui-ci), dans ce cas la boucle serait aussi probablement chargée en entier dans le cache, et donc le *row buffer* ne serait utilisé que lors du premier accès au bloc *persistent*, ce qui rend l'analyse de *persistence* inintéressante pour le *row buffer*.

Quant à la *persistence* du cache, on notera qu'un l-bloc *persistent* provoque un *Update* abstrait (au niveau de l'interprétation abstraite du row buffer) qui fait l'union des deux cas possibles (cache *hit* et cache *miss*), malgré que toutes les exécutions sauf la première soient des *hits* de cache. Cette imprécision pourrait être réduite en utilisant le déroulage de boucles.

3.2.3 Expérimentation

Nous avons expérimenté cette technique grâce à OTAWA, en utilisant la technique IPET. Le coût d'exécution des blocs de base ont été calculés grâce à la méthode des graphes d'exécution. Le cache a été analysé grâce à la méthode présentée dans la section 3, et le row buffer a été analysé grâce à la présente technique. Du point de vue matériel, les expérimentations ont été faites à l'aide d'un processeur superscalaire avec exécution dans l'ordre, doté d'un cache d'instructions de 512 octets associatif par ensemble à deux voies avec des blocs de 16 octets. La taille du cache simulé est volontairement faible, car les programmes utilisés - les benchmarks Malärdalen - sont petits. La taille du *row buffer* est fixée à 1024 octets, et les pénalités en cas de défaut de cache ou de *row buffer* sont fixées respectivement à 24 et 20 cycles. Ainsi, si un accès provoque un défaut de cache mais que la requête est ensuite satisfaite au niveau du *row buffer*, il y aura une latence de 24 cycles. Toutefois, si l'accès provoque un défaut de cache et aussi un défaut dans le row buffer, la pénalité sera de $24 + 20 = 44$ cycles.

Les résultats sont fournis dans la figure 3.8. Nous avons mesuré le ratio de l-blocs concernés par l'analyse du *row buffer* (c'est-à-dire les l-blocs dont la catégorie de cache n'est pas *Always Hit*). En moyenne, il y a 44% de l-blocs concernés. Nous avons aussi



(a) Blocs catégorisés et utilisés

(b) Impact du row buffer sur le WCET

FIGURE 3.8: Résultats d'analyse du row buffer

mesuré le pourcentage de chaque catégorie, en moyenne seulement 14% des l-blocs sont *Row buffer Not Classified*. Ces résultats sont montrés dans la figure 3.8(a) : pour chaque test, on voit la quantité de l-blocs concernés par l'analyse du row buffer (l-blocs utilisés), ainsi que la quantité de l-blocs catégorisés par l'analyse de *row buffer* (tous les l-blocs qui ne sont pas catégorisés *Row buffer Not Classified*).

Le WCET a aussi été mesuré, une fois en prenant en compte le *row buffer*, une fois sans le prendre en compte (c'est-à-dire faisant l'hypothèse pessimiste que chaque *miss* de cache provoquait un *miss* de *row buffer*). La figure 3.8(b) montre le rapport entre les deux WCET (sans et avec analyse de *row buffer*), et ce pour chaque test. En moyenne, le WCET estimé est réduit de 19 % par la prise en compte du *row buffer*.

Chapitre 4

Analyse partielle du cache

Dans cette partie, nous présentons notre approche pour l'analyse partielle du cache d'instructions. Nous parlons d'abord de l'approche de K. Patil et al. [70], puis nous proposons notre approche (qui est elle-même basée sur un travail de A. Rakib et al. dans [74]), et enfin nous validons notre approche par une expérimentation avec OTAWA.

4.1 Travaux de K. Patil et al. sur l'analyse de cache par composants

Dans [70], K. Patil et al. ont étendu la méthode définie dans [66, 67] pour qu'elle soit utilisable sur un programme composé de plusieurs modules. Cette approche comporte deux phases. Dans la première phase, on analyse chaque module quatre fois : chaque analyse correspond à un contexte potentiel différent (*scope)*. Cette phase permet d'obtenir des catégories temporaires, qui sont ensuite ajustées lors de la deuxième phase par une analyse globale du programme en tenant compte du contexte d'appel de chaque module. Cette approche est conçue principalement pour gagner du temps d'analyse, mais ne semble pas avoir été prévue pour faciliter le traitement de COTS, puisque la deuxième phase de l'analyse doit visiter le graphe de flot de contrôle de chaque composant.

Ceci est dommage car les COTS sont de plus en plus utilisés dans l'industrie, et il serait donc intéressant de pouvoir les prendre en compte grâce à l'analyse partielle. C'est pourquoi dans notre approche d'analyse partielle du cache nous essayons de remédier à cette limitation.

4.2 Approche générale de l'analyse partielle du cache

Cette analyse est basée sur l'analyse de cache présentée dans le chapitre précédent (analyses *May*, *Must*, version paramétrique de l'analyse *Persistence*, puis catégorisation) et produira le même type de résultats.

Le déroulement général de notre approche le suivant : on réalise l'analyse partielle d'un composant appelé `callee()`, qui retourne un résultat partiel d'analyse de cache.

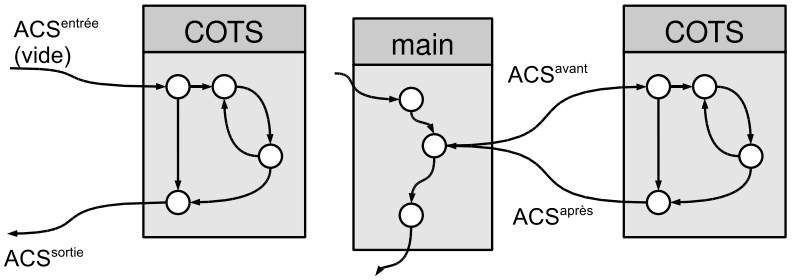


FIGURE 4.1: Clarification des notations

Ensuite, le programme principal (celui qui utilise le composant), appelé `main()`, peut être analysé sans avoir accès au code du composant, en utilisant uniquement le résultat partiel. Le résultat de l'ensemble de l'analyse sera une catégorisation de tous les l-blocs du programme (incluant le composant).

Lorsqu'on étudie un programme constitué de plusieurs composants, la notation ACS^{p}_{type} définie dans la section 2.4.3 peut devenir ambiguë, c'est pourquoi nous fournissons ici une notation plus spécifique. Tout d'abord, appelons $ACS^{(p,d,e)}_{type}$ l'ACS obtenue au point du programme p par une analyse *type* (*type* étant *may*, *must*, ou *pers*), en prenant comme état d'entrée du composant e (essentiellement $\top$ ou $\bot$) et comme point d'entrée d. Ceci est une généralisation de la notation ACS^{p}_{type}, en effet $ACS^{p}_{type} = ACS^{(p,d,\top)}_{type}$ où d est le point d'entrée du programme principal. Par convention, si p est un point du programme appartenant à un composant (que nous noterons C, par défaut) concerné par l'analyse partielle, alors ACS^{p}_{type} est $ACS^{(p,d,\top)}$, où d est le point d'entrée du composant, noté $entrée_C$ (contrairement au cas où p n'appartient pas à un composant, auquel cas le d choisi est le point d'entrée du programme principal, noté *entrée*).

Si on reste consistant avec ce principe, alors les notations $ACS^{entrée_C}_{type}$ et $ACS^{sortie_C}_{type}$ désignent les ACS à l'entrée et à la sortie du composant C, obtenues grâce à une analyse indépendante du composant (il est à noter que $ACS^{entrée_C}_{type}$ sera toujours égal à $\top$ par définition, mais nous l'avons quand même défini ici par souci d'exhaustivité). Les points du programme avant et après l'appel au composant C sont nommés respectivement $avant_C$ et $après_C$, ces points du programme sont équivalent à $entrée_C$ et $sortie_C$ mais ils ne sont pas considérés comme faisant partie de C. Donc, les notations $ACS^{avant_C}_{type}$ (resp. $ACS^{après_C}_{type}$) désignent les ACS au moment de l'appel au composant (resp. du retour du composant) tels qu'ils seraient calculés par une analyse complète de la tâche (programme principal et composant), c'est-à-dire en prenant l'entrée principale du programme comme point de départ d'analyse. La figure 4.1 présente un exemple pour clarifier ces notations.

Nous utiliserons ces conventions de nommage tout au long de la présentation des méthodes d'analyses partielles.

Puisque notre méthode s'appuie sur l'algorithme présenté par A. Rakib et al. dans [74], nous décrivons d'abord ces travaux avant de présenter notre méthode.

4.2.1 Les travaux de A. Rakib et al. sur le dommage de cache

Dans l'article [74], A. Rakib et al. adaptent la méthode de prédiction de comportement du cache d'instructions présentée par M. Alt et al. dans [34, 32, 87, 3] pour permettre l'analyse d'un exécutable construit à partir de plusieurs composants. Dans cette approche, chaque fichier objet (correspondant à un composant) est analysé pour produire un résultat partiel. Ensuite, ces résultats partiels sont composés pour obtenir un WCET final.

Cette méthode tente de résoudre deux problèmes :

- Lorsqu'on réalise l'analyse partielle de `callee()`, l'adresse de base du composant n'est pas connue. Or, cette adresse de base détermine quelles lignes de cache sont vieillies et quels blocs de cache sont chargés lors de l'exécution du composant. Pour contourner ce problème, on force *l'éditeur de lien* à placer le composant à une adresse mémoire équivalente (du point de vue du comportement du cache) à celle utilisée lors de l'analyse partielle. Concrètement, cela veut dire que l'adresse d'implantation doit être équivalente à l'adresse utilisée lors de l'analyse partielle, modulo la taille du cache.
- Lors de l'exécution de `callee()`, cette fonction vieillit et/ou remplace des blocs de cache du programme principal, cela affecte la catégorisation des l-blocs du programme principal. Ceci sera modélisé par une fonction de dommage de cache, pour chaque analyse de cache (*May*, *Must*, et *Persistence*). La fonction de dommage est définie par $\widetilde{ACS}_{type}^{après_C} = damage(ACS_{type}^{avant_C})$, où *type* est *may* ou *must* (la *persistence* n'est pas gérée dans ces travaux), et où $\widetilde{ACS}_{type}^{après_C} \supseteq ACS_{type}^{après_C}$ (qui nous assure que nous avons au pire une surestimation). Elle fournit, pour chaque analyse de cache, l'ACS après l'appel du composant en fonction de l'ACS avant l'appel du composant.

La fonction *dommage* modélise, pour chaque analyse, l'effet de l'appel au composant sur l'ACS du programme principal. Pour y parvenir, il faut tenir compte (1) des blocs du programme principal qui vont être remplacés par des blocs du composant, et (2) des blocs du composant qui seront insérés dans le cache lors de l'exécution du composant. Par conséquent, la construction de la fonction *dommage* nécessite au préalable le calcul de deux éléments distincts :

- Le *vieillissement* pour chaque ensemble de cache l, qu'on notera $DMG_{type}^{sortie_C}(l)$. Le vieillissement est représenté par un entier compris dans l'ensemble $[0..A]$ (A est le nombre de voies du cache associatif par ensemble), cet entier représente la borne supérieure (pour l'analyse *must*) ou bien la borne inférieure (pour l'analyse *may*) du vieillissement possible d'un bloc de cache quelconque appartenant à l'ensemble l pendant l'exécution du composant. Le but du DMG est de savoir quels blocs présents dans l'ACS d'entrée sont toujours présents en sortie du composant (et à quels âges).
- Les *blocs insérés* pour chaque ensemble de cache l, qu'on notera $ACS_{type}^{(sortie_C, entrée_C, \emptyset)}(l)$. Ceci permet de savoir quels sont les blocs de cache appartenant au composant qui se retrouvent dans l'ACS au retour de celui-ci, quel que soit le contexte d'appel.

$$\forall l\, DMG' = Update_{must}(DMG, lb) / DMG'(l) =$$
$$\begin{cases} DMG(l) & si\ ligne_{lb} \neq l \\ DMG(l) & si\ ligne_{lb} = l \wedge bloc_{lb} \in ACS_{must}^{lb, entrée_C, \emptyset}(l) \\ DMG(l) \oplus 1 & sinon \end{cases}$$
$$\forall l\, DMG' = Join_{must}(DMG1, DMG2) /$$
$$DMG'(l) = MAX(DMG1(l), DMG2(l))$$

FIGURE 4.2: Update et Join de l'analyse de vieillissement (*must*)

$$\forall l\, DMG' = Update_{may}(DMG, lb) / DMG'(l) =$$
$$\begin{cases} DMG(l) & if\ ligne_{lb} \neq l \\ DMG(l) & if\ ligne_{lb} = l \wedge bloc_{lb} \in ACS_{may}^{lb, entrée_C, \emptyset}(l) \\ DMG(l) \oplus 1 & sinon \end{cases}$$
$$\forall l\, DMG' = J_{may}(DMG1, DMG2) /$$
$$DMG'(l) = MIN(DMG1(l), DMG2(l))$$

FIGURE 4.3: Update et Join de l'analyse de vieillissement (*may*)

Comme précisé plus haut dans cette section, les blocs insérés ($ACS_{type}^{sortie_C, entrée_C, \emptyset}(l)$) peuvent être calculés par une analyse indépendante du composant, en utilisant le *May*, *le Must*, et le *Pers* traditionnels, et en prenant l'ACS vide comme ACS d'entrée du composant. Toutefois, le vieillissement est calculé par une analyse spécifique.

Le vieillissement est calculé indépendamment pour chaque ensemble de cache par une interprétation abstraite dont le domaine est un entier dans l'ensemble $[0..A]$. Le domaine est le même pour le vieillissement *may* et *must*, mais les fonctions *Update* et *Join* sont différentes. Les fonctions *Update* et *Join* des analyses de vieillissement *must* et *may* sont données, respectivement, dans les figures 4.2 et 4.3. Dans ces figures, l'opérateur $\oplus$ est défini, tel que $x \oplus y = MAX(A,\ x+y)$. On notera que l'analyse de vieillissement a besoin de l'ACS *must*, c'est pourquoi l'analyse indépendante du composant (utile de toute façon pour calculer les blocs insérés) doit être réalisée avant l'analyse de vieillissement.

La fonction *dommage* du *must* est définie dans la figure 4.4, en combinant le vieillissement des blocs existants (en utilisant $ACS_{must}^{avant_C}\ DMG_{must}^{sortie_C}$) et le rajout des blocs insérés (en utilisant $ACS_{must}^{sortie_C}$). La fonction *damage* du *may* est définie selon le même principe sur la figure 4.5.

$$\forall l,\ \forall a \in [0..A[\ \widetilde{ACS}_{must}^{après_C} =$$
$$Damage(ACS_{must}^{avant_C}, ACS_{must}^{sortie_C}, DMG_{must}^{sortie_C}) / ACS_{must}^{après_C}(l, a) =$$
$$\begin{cases} ACS_{must}^{sortie_C}(l, a) & \forall a \in [0, j-1] \\ ACS_{must}^{avant_C}(l, a-j) \backslash blocks(ACS_{must}^{sortie_C}(l)) & \forall a \in [j, A-1] \\ où\ j = DMG_{must}^{sortie_C}(l) & \end{cases}$$

FIGURE 4.4: Fonction *dommage (must)*

$$\forall l,\ \forall a \in [0..A[\ \widetilde{ACS}^{après_C}_{may} =$$
$$Damage(ACS^{avant_C}_{may}, ACS^{sortie_C}_{may}, DMG^{sortie_C}_{may})\,/ACS^{après_C}_{may}(l,a) =$$
$$\begin{cases} ACS^{sortie_C}_{may}(l,a) & \forall a \in [0, j-1] \\ ACS^{avant_C}_{may}(l, a-j) \backslash blocks(ACS^{sortie_C}_{may}(l)) & \forall a \in [j, A-1] \\ où\ j = DMG^{sortie_C}_{may}(l) \end{cases}$$

FIGURE 4.5: Fonction *dommage (may)*

4.2.2 Notre méthode d'analyse partielle

Notre méthode utilise des éléments de l'analyse partielle du cache proposée par A. Rakib et al., mais cette dernière comporte un certain nombre de limitations, que nous nous sommes efforcé de corriger dans notre approche :

- Le dommage de cache proposé par A. Rakib et al. pourrait être nettement amélioré du point de vue de la précision pour le *may* et le *must*. De plus, ce dommage de cache ne prend pas du tout en compte la *persistence*. Nous avons introduit une version améliorée du dommage de cache, plus précis et capable de gérer la *persistence*.
- La fonction `callee()` contient des l-blocs qui ont besoin d'être catégorisés. Malheureusement, la catégorie de ces l-blocs n'est pas constante, elle dépend du contexte, c'est-à-dire des ACS lors de l'appel de `callee()`. Comme l'analyse de A. Rakib et al. ne prévoit pas de mécanisme pour gérer cela, nous avons introduit dans [6] une fonction *résumé* (une seule fonction est nécessaire pour les trois analyses *May*, *Must*, et *Persistence*). Cette fonction est définie par $condcat = résumé(lblock)$, où *condcat* est une Catégorie Conditionnelle (CC), c'est-à-dire une fonction de signature $CC : ACS^{entrée_C}_{pers} \times ACS^{entrée_C}_{must} \times ACS^{entrée_C}_{may} \rightarrow AH \,|\, AM \,|\, PS \,|\, NC$. En d'autres termes, la fonction *résumé* permet d'avoir, en fonction des ACS avant l'appel à `callee()`, une catégorie pour chacun de ses l-blocs.
- La contrainte concernant l'adresse d'implantation du composant pourrait être allégée. En effet, il nous suffit de faire en sorte que la position relative des l-blocs de `callee()` par rapport aux frontières de blocs de cache soient les mêmes lors de l'analyse partielle et lors de l'utilisation du résultat partiel. En d'autre termes, pour chaque l-bloc *lb* de `callee()`, la valeur $lb_{addr}\ mod\ taillebloc$ (où *taillebloc* est la taille du bloc de cache) devra être la même lors de l'analyse partielle et lors de la composition. Grâce à cette propriété, la correspondance entre les blocs de cache référencés dans le résultat partiel et les blocs de cache réels pourra être réalisée par de simples rotations des fonctions. Cette contrainte (modulo taille du bloc) est plus faible que celle de A. Rakib et al. (taille du cache) et évite un trop gros gaspillage de mémoire entre les composants (au plus $taille_{bloc\,de\,cache} - 1$ octets).

Pour notre analyse, le résultat partiel de l'analyse de cache pour un composant donné est donc défini par ses fonctions *dommage* et *résumé*. La création et l'application de ces fonctions seront vues en détail lors des prochaines sections. Lorsqu'on veut utiliser ce résultat partiel en présence d'un programme principal qui utilise le composant, l'instanciation du

résultat partiel se fait en deux étapes, sans avoir besoin d'accéder au code de `callee()` :

- On réalise l'analyse du cache d'instructions sur le programme principal, tel qu'on l'aurait fait avec une analyse traditionnelle (monolithique), à ceci près qu'on remplace l'appel à `callee()` par un bloc de base virtuel dont la fonction *update* est en fait la fonction *transfert* du composant. A la fin de cette étape, on dispose des ACS avant et après chaque bloc de base du programme principal.
- Une fois qu'on a les ACS en tout point du programme principal, on connaît la valeur des ACS lors de l'appel à `callee()`, et ce pour tous les sites d'appel. Ceci nous permet donc, pour chaque site d'appel, d'appliquer la fonction *résumé* afin d'obtenir la catégorisation des l-blocs de `callee()`.

A la fin de cette étape de composition, on dispose de la catégorisation des l-blocs de toute la tâche (programme principal, et composant) exactement comme avec une analyse de cache monolithique.

4.2.3 Nos travaux sur les fonction de dommage de cache

Nous avons réalisé deux améliorations de la fonction de dommage de cache que nous présentons dans les deux prochaines sous-sections. Lors de l'explication de cette analyse, il sera fréquemment nécessaire de raisonner à l'aide de comparaisons d'âges de blocs. Pour les besoins de ces comparaisons, dans un ACS, un bloc qui est absent sera considéré comme ayant pour âge l'entier A (degré d'associativité du cache) et est donc considéré comme supérieur à tout autre âge (réel). La seule exception à ceci est la *persistence*, dans laquelle lorsqu'un bloc peut avoir été mis dans le cache et ensuite supprimé, ce bloc possédera l'âge $l_\top$. L'ordre de comparaison pour la *persistence* est défini comme suit : $age\, A < age\, 0 < ... < age\, (A-1) < age\, l_\top$. Ainsi, l'âge A dans la *persistence* (correspondant à un bloc absent qui n'est jamais entré dans le cache) est le plus petit élément, tan disque l'âge $l_\top$ (correspondant à un bloc absent mais qui a pu être dans le cache auparavant) correspond à l'âge le plus vieux. Cet ordre pour la *persistence* sera utile dans la définition de notre fonction de dommage, et découle de la décomposition du *Join* de la *persistence*. En effet, $Join_{pers}(ACS1, ACS2)$ peut être exprimé comme l'ACS pour laquelle l'âge de chaque bloc *cb* est le MAX (dans l'ordre donné précédemment) des âges des blocs *cb* dans $ACS1$ et $ACS2$.

4.2.3.1 Analyse de vieillissement améliorée

Afin de détecter les cas qui peuvent être améliorés dans la gestion du dommage de cache de A. Rakib et al. [74], analysons les informations de vieillissement de cache sur la figure 4.6(a) (pour des raisons de simplicité, nous avons fait en sorte que dans cet exemple, $1 \notin ACS^{sortie_C}_{must}(l)$, pour que $ACS^{après_C}_{must}(l)$ ne dépende que du vieillissement, et non des blocs insérés). La cause d'imprécision la plus flagrante est le fait que le dommage de cache ne fait pas de différence entre les différents blocs projetés sur un même ensemble. Ce problème est bien illustré sur la figure : on voit que le vieillissement est de 2 (pour

la ligne considérée). Puisque on est en présence d'un cache associatif à 2 voies, et que l'information de vieillissement est calculée globalement pour chaque ensemble, tous les blocs de cache projetés sur cet ensemble seront considérés comme éliminés du cache après l'exécution du composant. Ceci provoque du pessimisme : si on s'intéresse au bloc 1, d'après l'information de vieillissement calculée, ce bloc est éliminé du cache, alors que ce n'est pas vrai. En effet, soit on passe par le chemin de gauche, auquel cas le bloc n'est pas vieilli du tout (car le l-bloc sur le chemin de gauche n'appartient pas à l'ensemble de cache courant de l'analyse), soit on passe par le chemin de droite auquel cas le bloc est vieilli une fois seulement (par le chargement du bloc 2). Le pire cas de vieillissement de ce bloc est donc 1, et non 2, mais ceci ne peut être détecté que si on réalise l'analyse de vieillissement bloc par bloc, et non ensemble par ensemble.

Pour réduire ce pessimisme, nos informations de vieillissement associent un entier dans $[0..A]$ à chaque bloc de cache (et non à chaque ensemble). Un entier supplémentaire est réservé pour représenter « tout bloc n'apparaissant pas dans le composant ». En effet, ces types de blocs seront la plupart du temps tous vieillis de la même manière donc le fait de les traiter ensemble n'introduit que peu de pessimisme, et ceci permettra donc d'exprimer dans le vieillissement une information de type « tout bloc extérieur au composant sera vieilli au maximum 2 fois ». On notera donc $[x \rightarrow y, ...]$ l'information de vieillissement qui associe l'âge y au bloc de cache x. Nous proposons de nouvelles fonctions *Join* et *Update* pour prendre en compte ces modifications, définies dans la figure 4.7 pour l'analyse *must*, et dans la figure 4.8 pour l'analyse *may*.

Le vieillissement d'un bloc de cache spécifique cb présent à l'entrée du composant se fera par l'opération : $cb \oplus DMG^{sortie_C}[cb]$.

Pour le *must* et le *m*ay, l'opérateur $\oplus$ est défini tel que si $cb + DMG^{après_C}[cb] \geq A$ alors $cb \oplus DMG^{sortie_C}[cb] = A$, dans le cas contraire $cb + DMG^{sortie_C}[cb] = cb \oplus DMG^{sortie_C}[cb]$.

Pour la *persistence*, l'opérateur $\oplus$ est défini de telle sorte que si $cb = A$ alors $cb \oplus DMG^{après_C}[cb] = A$, sinon si $cb + DMG^{sortie_C}[cb] \geq A$ alors $cb \oplus DMG^{sortie_C}[cb] = l_\top$, sinon $cb + DMG^{sortie_C}[cb] = cb \oplus DMG^{sortie_C}[cb]$. Cette modification est nécessaire, en effet si un bloc de cache présent à l'entrée du composant est suffisamment vieilli pour se retrouver hors du cache, alors son nouvel âge doit être $l_\top$, conformément à la définition de la *persistence*. Ceci implique qu'il faut traiter spécifiquement le cas où le bloc cb n'est pas dans le cache à l'entrée du composant : alors il n'est pas affecté par le vieillissement (le bloc n'étant pas présent il ne peut pas être vieilli et il ne doit donc pas avoir l'âge $l_\top$ à la sortie du composant).

4.2.3.2 Fonction de dommage de cache améliorée

Maintenant qu'on a spécifié les analyses de vieillissement améliorées, il devient possible de donner, en fonction de celles-ci, l'expression des fonctions de dommage[1] de cache améliorées pour le *must* et le *may* (le cas du *pers* sera traité plus bas).

1. nous avons choisi de donner notre propre définition de la fonction *dommage,* plus complète (notamment incluant le *pers*), et sur laquelle il nous semble qu'il est plus facile de raisonner.

Soit ACS^{avant}_{type} (où *type* est *must* ou *may*) l'ACS calculé à l'appel d'un composant. Conformément à la notation exposée précédemment, on peut appeler $ACS^{après_C}_{type}$ l'ACS qui serait calculée par une analyse classique de cache pratiquée sur le composant en prenant $ACS^{avant_C}_{type}$ pour état initial. L'ACS $\widetilde{ACS}^{après_C}_{type}$ est une estimation, et donc doit être plus grand ou égal (dans l'ordre du treillis) que $ACS^{après_C}_{type}$. $ACS^{après_C}_{type}$ est bien sûr lui aussi une estimation, correspondant aux états de caches concrets possibles durant l'exécution, mais cette estimation est en général plus précise que $\widetilde{ACS}^{après}_{type}$). Appelons $dommage^*_{type}$ la fonction de dommage de cache idéale qui, à partir d'un ACS^{avant}_{type} fournit, $ACS^{après}_{type}$.

Soit $ACS^{sortie_C,entrée_C,\emptyset}_{type}$ l'ACS calculée par une analyse du composant en prenant $\emptyset$ comme état initial (c'est à dire les blocs insérés), et soit $DMG^{sortie_C}_{type}$ l'information de vieillissement. Intéressons-nous d'abord au cas du *must* pour lequel nous devons montrer deux propriétés :

1. Pour chaque bloc de cache *cb*, on a $ACS^{sortie_C,entrée_C,\emptyset}_{must}[cb] \geq ACS^{sortie_C}_{must}[cb]$, ce qui revient à dire que l'âge de *cb* calculé dans les blocs insérés est forcément supérieur à l'âge qui aurait été trouvé à la sortie du composant si on avait réalisé une analyse non-partielle. Rappelons que, pour les besoins de la comparaison des âges dans les ACS, nous considérons que l'âge d'un bloc qui n'est pas dans le cache est supérieur à tous les autres âges.
2. Pour chaque bloc de cache *cb* présent dans $ACS^{avant_C}_{must}$, son âge après application des informations de vieillissement ($ACS^{avant_C}_{must}[cb] \oplus DMG^{sortie_C}_{must}[cb]$) est forcément au moins aussi vieux que l'âge réel de *cb* dans $ACS^{après_C}_{must}$, et ce par définition de l'information de vieillissement.

Grâce à (1) et (2), on peut déduire que $MIN(ACS^{sortie_C,entrée_C,\emptyset}_{must}[cb], ACS^{avant_C}_{must}[cb] \oplus DMG^{sortie_C}_{must}[cb])$ est forcément supérieur ou égal à $ACS^{après_C}_{must}[cb]$. Ceci nous amène directement à la définition de la fonction de dommage de cache du *must*, présentée dans la figure 4.9(a) (l'opération $\cap$ pour le *must* correspond au *MIN* bloc de cache par bloc de cache, en considérant qu'un bloc absent possède l'âge le plus vieux).

Considérons maintenant le cas du *may*. Pour chaque bloc de cache *cb*, divisons le composant en deux ensembles de chemins (allant de l'entrée du composant vers sa sortie), appelons ces deux composants (virtuels) $C1$ et $C2$:

1. L'ensemble des chemins ne chargeant pas *cb*. L'âge de *cb* obtenu par application de l'information de vieillissement ($ACS^{avant_C}_{may}[cb] \oplus DMG^{sortie_C}_{may}[cb]$) est forcément plus jeune (ou égal) à $ACS^{après_{C1}}_{may}[cb]$, par définition.
2. L'ensemble des chemins chargeant *cb* (et ce, même si ce bloc doit être supprimé du cache avant l'arrivée à la sortie du composant). L'âge de *cb* dans $ACS^{sortie_C,entrée_C,\emptyset}_{may}$ est égal à $ACS^{après_{C2}}_{may}[cb]$, en effet, puisque tous les chemins de $C2$ font référence à *cb*, l'âge original de *cb* (ou même sa présence) dans $ACS^{avant_{C2}}_{may}$ importe peu.

Puisque $C1$ et $C2$ ne sont qu'une subdivision de C, on sait que $ACS^{après_{C1}}_{may} \cup ACS^{après_{C2}}_{may}$ est une abstraction valide pour l'ensemble des états concrets qui arriveront au point $après_C$, et d'après (1) et (2), $ACS^{sortie_C,entrée_C,\emptyset}_{may} \cup (ACS^{avant_C}_{may}[cb] \oplus DMG^{sortie_C}_{may}[cb])$ en est une aussi

(bien que moins précise). L'union (Join) du *may* étant en fait le *MIN* bloc de cache par bloc de cache (en considérant qu'un bloc absent possède l'âge le plus vieux), la fonction de dommage de cache du *may* a la même structure que celle du *must*, et est présentée dans la figure 4.9(b).

Étudions le cas de la *persistence*. Pour chaque bloc de cache *cb*, on peut diviser le composant en $C1$ et $C2$ comme pour le *may*, et l'âge de *cb* obtenu par le vieillissement est forcément plus jeune (ou égal) à $ACS^{après_{C1}}_{pers}[cb]$. Le vieillissement utilisé pour la *persistence* est le même que pour le *must* ($DMG^{sortie_C}_{must}[cb]$), car tous deux prennent en compte l'âge le plus vieux possible. Également, l'âge de *cb* dans $ACS^{sortie_C, entrée_C, \emptyset}_{pers}$ est égal à $ACS^{après_{C2}}_{pers}[cb]$. Pour les mêmes raisons que pour le cas du *may*, $ACS^{sortie_C, entrée_C, \emptyset}_{pers} \cup (ACS^{avant_C}_{pers}[cb] \oplus DMG^{sortie_C}_{must}[cb])$ est une abstraction valide pour l'ensemble des états concrets qui peuvent arriver au point $après_C$, ce qui permet d'obtenir la fonction de dommage du *pers*, présentée dans la figure 4.9(c).

4.2.3.3 Analyse de vieillissement finale

Cette amélioration résout le premier problème de pessimisme, exposé dans la figure 4.6(a). Toutefois, il reste un problème, que nous avons illustré grâce à l'exemple de la figure 4.6(b). Nous voyons que $ACS^{avant}_{must}(l) = [3, 1]$ (l étant l'ensemble de cache courant de l'analyse), et que $DMG^{sortie}(l) = [1 \rightarrow 1, 2 \rightarrow 0]$ puisque le bloc 1 est vieilli au pire une fois (à cause du chargement du bloc de cache 2). Par conséquent, $ACS^{après}_{must}$, c'est-à-dire le résultat de l'application du vieillissement sur ACS^{avant}_{must} est égal à [3,]. Le bloc de cache 1 est donc éliminé. Ceci est pessimiste : si le bloc de cache 1 est présent à l'entrée du composant, on peut en déduire qu'il est aussi présent à la sortie. En effet, si le chemin de gauche est pris, le bloc 1 est chargé puis vieilli une fois, alors que dans le chemin de droite ce bloc n'est pas vieilli du tout. Dans les deux cas, le bloc 1 est présent à la sortie, ce qui n'est pas détecté par notre analyse. Le problème vient du fait que l'entier représentant le vieillissement d'un bloc (et qui est donc ajouté à l'âge original de ce bloc à l'entrée du composant) est calculé pour tous les chemins à l'intérieur du composant, même pour les chemins pour lesquels il n'est pas approprié de vieillir l'âge du bloc de cette manière.

Notre solution consiste à séparer l'information de vieillissement du *must* en deux parties (l'information de vieillissement du *may* n'est pas modifiée car l'effet présenté dans la figure ne s'applique qu'au *must*), DMG_{in} et DMG_{out}, c'est-à-dire que chaque bloc de cache *cb* se verra associé à deux entiers au lieu d'un ($DMG_{in}(l, cb)$ et $DMG_{out}(l, cb)$). Le $DMG_{in}(l, cb)$est le vieillissement calculé par une analyse ne prenant en compte que les chemins contenant une référence à *cb*. Inversement, le $DMG_{out}(l, cb)$ est calculé en prenant en compte les chemins ne contenant *pas* de référence à *cb*. La valeur spéciale $\emptyset$ sera désignée pour représenter l'absence de tout chemin analysable. Par exemple, considérons le cas du $DMG_{out}(l, cb)$: si absolument tous les chemins à l'intérieur du composant référencent *cb*, alors $DMG_{out}(l, cb) = \emptyset$.

Par exemple, dans la figure 4.6(b), on a $DMG^{sortie}_{in}(l, 1) = 1$ et $DMG^{sortie}_{out}(l, 1) = 0$, ces

deux entiers représentent le vieillissement du bloc 1 respectivement sur le chemin de gauche et de droite. On peut noter que le DMG_{in} est aisément déductible à partir de la *persistence*, ce qui fait qu'il n'y a que le DMG_{out} à calculer via interprétation abstraite. Ceci veut dire que notre séparation du DMG en deux ne nécessite pas d'alourdir l'analyse de vieillissement par une interprétation abstraite supplémentaire. La figure 4.10(a) montre les fonctions *Update* et *Join* pour le calcul du DMG_{out}. L'état de l'analyse du vieillissement est initialisé à l'entrée du composant comme suit : $\forall l, cb : DMG_{in}(l, cb) = \emptyset \wedge DMG_{out}(l, cb) = 0$.

4.2.3.4 La fonction de dommage finale (*Must*)

La définition du vieillissement de cache du *must* ayant été modifiée, il est nécessaire de refléter ces modifications dans la fonction de dommage de cache du *must*. Il est aussi nécessaire de refléter les modifications dans la fonction de dommage de cache de la *persistence* étant donné que celle-ci utilise le vieillissement du *must*, ceci est fait dans la prochaine sous-section.

Commençons d'abord par exprimer la nouvelle façon d'appliquer l'information de vieillissement. Soit un bloc de cache *cb*, d'âge $ACS_{must}^{avant_C}[cb]$ à l'entrée du composant. Pour savoir son âge $ACS_{must}^{après_C}[cb]$ à la sortie du composant, nous prendrons l'âge maximum (pire cas) de ces deux scénarios :

1. L'exécution passe par un chemin dans le composant qui ne fait pas référence à *cb*, dans ce cas l'âge de *cb* en sortie sera $ACS_{must}^{avant_C}[cb] \oplus DMG_{out}^{sortie}[cb]$, ce cas est similaire à l'application du vieillissement traditionnel présenté dans la section 4.2.3.2.
2. L'exécution passe par un chemin dans le composant qui fait référence à *cb*, dans ce cas l'âge de *cb* en sortie sera $DMG_{in}^{sortie}[cb]$ et non $ACS_{must}^{avant_C}[cb] \oplus DMG_{out}^{sortie}[cb]$, car l'âge original du bloc *cb* ne doit pas être pris en compte, puisqu'il sera rafraîchi. C'est précisément ce point qui fait que cette nouvelle analyse de vieillissement est plus précise.

L'application du vieillissement au bloc *cb* lui donne donc l'âge suivant en sortie :

$$MAX(DMG_{in}^{sortie}[cb],\ ACS_{must}^{avant_C}[cb] \oplus DMG_{out}^{sortie}[cb]).$$

On pourrait en déduire, comme dans 4.2.3.2, que l'âge final (après application du dommage complet avec les blocs insérés) est :

$$MIN(ACS_{must}^{sortie_C, entrée_C, \emptyset}[cb], MAX(DMG_{in}^{sortie}[cb],\ ACS_{must}^{avant_C}[cb] \oplus DMG_{out}^{sortie}[cb]))$$

Toutefois, ceci n'est pas nécessaire. En effet, si un bloc *cb* est mis dans le cache par le composant quel que soit le contexte (c'est-à-dire que $ACS_{must}^{sortie_C, entrée_C, \emptyset}[cb] \neq A$), alors ceci signifie que tous les chemins passent par *cb*, et que donc $DMG_{out}^{sortie} = \emptyset$. Puisque *cb* sera forcément rafraîchi par le passage dans le composant, alors son âge maximum après le composant est donné par $DMG_{in}^{sortie}[cb]$. En résumé, si $DMG_{out}^{sortie}[cb] = \emptyset$ alors $\widetilde{ACS}_{must}^{après_C}[cb] = DMG_{in}^{sortie}[cb]$, sinon $\widetilde{ACS}_{must}^{après_C}[cb] = MAX(DMG_{in}^{sortie}[cb],\ ACS_{must}^{avant_C}[cb] \oplus DMG_{out}^{sortie}[cb])$

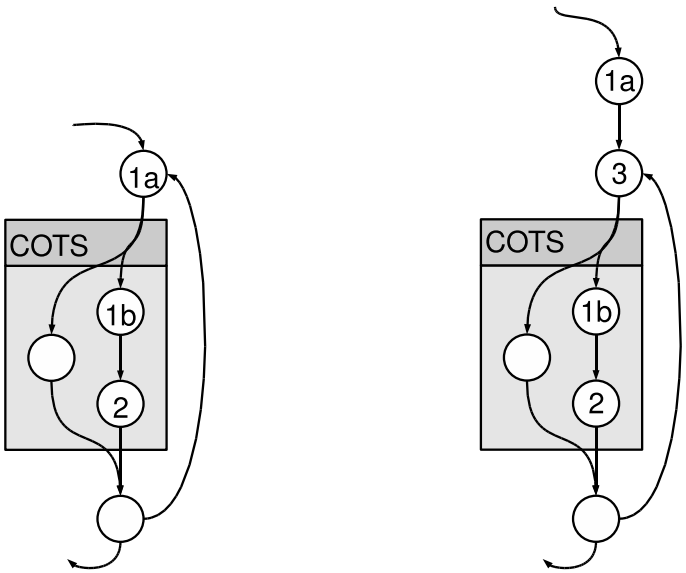
Cette distinction est implémentée par notre nouvelle fonction de dommage décrite figure 4.11(a). Par conséquent, la fonction de dommage est uniquement dépendante de $DMG^{exit}_{in/out}$, et il n'y a plus besoin de calculer l'ensemble des blocs insérés, $ACS^{sortie_C,entrée_C,\emptyset}_{must}$.

4.2.3.5 La fonction de dommage finale (*Persistence*)

Pour la *persistence*, les blocs vieillissent de manière similaire dans les ACS de la *persistence* et du *must*. Ainsi les informations de vieillissement calculées pour le *must* permettent aussi de traiter la *persistence*. Le dommage de cache pour la *persistence* est appliqué comme suit : pour chaque bloc de cache *cb* donné il y a deux possibilités. Soit le bloc de cache n'était pas dans l'ACS de *persistence* au début du composant, donc les chemins du composant qui ne font pas référence au bloc ne pourront pas affecter son statut dans l'ACS, et donc son âge en sortie sera déterminé uniquement par $DMG^{sortie}_{in}(l, cb)$ (qui est en fait équivalent à l'âge qui aurait été donné par ACS^{sortie}_{pers}). Soit le bloc de cache était dans l'ACS de *persistence*, et donc il faut vieillir le bloc à l'aide de $DMG^{sortie}_{in}(l, cb)$ et de $DMG^{sortie}_{out}(l, cb)$, et prendre le pire des cas. Enfin, si $DMG^{sortie}_{out}(l, cb) = \emptyset$ ceci veut dire que tous les chemins du composant référencent *cb*, donc le nouvel âge de *cb* est $DMG^{sortie}_{in}(l, cb)$, ceci est le seul cas où le bloc peut être *rajeuni* par l'exécution du composant. La formule exacte du dommage en fonction de DMG^{sortie}_{in} et DMG^{sortie}_{out} est donnée dans la figure 4.11. Malgré le fait qu'elle ait la même expression que la fonction de dommage pour le *must*, ce n'est pas la même fonction, en effet la fonction MAX et l'opérateur $\oplus$ n'ont pas la même signification.

4.2.3.6 Fonction de dommage et *persistence* paramétrique

Il est logique de se demander comment le dommage de cache sera appliqué dans le cas de la *persistence* paramétrique, telle que décrite dans le chapitre 3. Nous rappelons que l'ACS de la *persistence* paramétrique est une pile de n éléments, chaque élément étant une ACS de *persistence* classique et correspondant à un niveau d'imbrication de boucle. Chaque composant est muni d'un seul arc d'entrée, et d'un seul arc de sortie. Ceci implique que si une tête de boucle est dans le composant, alors tous les blocs de base de la boucle sont aussi dans le composant. En effet, soit H une tête de boucle dans le composant , et soit B un bloc de base appartenant à la boucle et étant situé à l'extérieur du composant, alors il existe un chemin $(B \rightarrow ... \rightarrow E_C \rightarrow ... \rightarrow R \rightarrow H)$, E_C étant l'unique prédécesseur de l'arc d'entrée du composant, et (R, H) étant un arc retour de la boucle. Soit E un bloc de base situé à l'extérieur du composant et n'appartenant pas à la boucle tel qu'il existe un chemin $(E \rightarrow ... \rightarrow E_C \rightarrow ... \rightarrow P \rightarrow H)$. Ce bloc E existe forcément, et (P, H) est un arc d'entrée de la boucle. Or, puisque le chemin $(B \rightarrow ... \rightarrow E_C \rightarrow ... \rightarrow R \rightarrow H)$ existe, le sous-chemin $(E_C \rightarrow ... \rightarrow R \rightarrow H)$ aussi, et puisque le chemin $(E \rightarrow ... \rightarrow E_C \rightarrow ... \rightarrow P \rightarrow H)$ existe, alors le sous-chemin $(E \rightarrow ... \rightarrow E_C)$ existe aussi. Donc, le chemin $(E \rightarrow ... \rightarrow E_C \rightarrow ... \rightarrow R \rightarrow H)$, qui est une simple concaténation des deux chemins précédents, doit exister. Or, l'existence de ce chemin revient à affirmer que E est dans la boucle. On aboutit donc à une contradiction,



(a) Problème 1 (b) Problème 2

FIGURE 4.6: Vieillissement de cache et pessimisme

$$\forall l\, DMG' = Update_{must}(DMG, lb) / DMG'(l, cb) =$$

$$\begin{cases} 0 & si\ ligne_{lb} = l \wedge bloc_{lb} = cb \\ DMG(l, cb) & si\ ligne_{lb} \neq l \\ DMG(l, cb) & si\ ligne_{lb} = l \wedge bloc_{lb} \in ACS^{lb}_{must}(l) \\ DMG(l, cb) \oplus 1 & sinon \end{cases}$$

$$\forall l\, DMG' = Join_{must}(DMG1, DMG2)\ /$$
$$DMG'(l) = MAX(DMG1(l), DMG2(l))$$

FIGURE 4.7: Vieillissement de cache, étape intermédiaire (*must*)

ce qui montre qu'il n'est pas possible d'avoir une boucle dont la tête est dans le composant et qui possède des blocs à l'extérieur du composant dans le cas où le composant n'a qu'un seul arc d'entrée et de sortie.

Cette propriété permet d'affirmer qu'il y a correspondance entre les éléments de l'ACS de *persistence* paramétrique avant et après un appel à un composant. Pour cette raison, l'application du dommage de cache pour la *persistence* paramétrique revient à appliquer le dommage individuellement sur chaque élément de l'ACS. Si l'ACS ne contient aucun élément (c'est-à-dire si l'appel au composant n'est dans aucune boucle), alors le dommage de cache de la *persistence* paramétrique peut être ignoré.

4.2.4 Fonction *résumé*

Dans cette partie nous décrivons la fonction *résumé,* d'abord pour le cas du *must*, du *may*, et de la *persistence* non paramétrique. Ensuite nous étudierons le cas de la *persistence* paramétrique.

$$\forall l, cb\, DMG' = Update_{may}(DMG, lb)/DMG'(l, cb) =$$
$$\begin{cases} 0 & si\, ligne_{lb} = l \wedge bloc_{lb} = cb \\ DMG(l, cb) & si\, ligne_{lb} \neq l \\ DMG(l, cb) & si\, ligne_{lb} = l \wedge bloc_{lb} \in ACS^{lb}_{may}(l) \\ DMG(l, cb) \oplus 1 & sinon \end{cases}$$
$$\forall l\, DMG' = Join_{may}(DMG1, DMG2)\,/$$
$$DMG'(l) = MIN(DMG1(l), DMG2(l))$$

FIGURE 4.8: Vieillissement de cache, étape intermédiaire (*may*)

$$\begin{array}{rcl} \forall l, cb & \widetilde{ACS}^{après_C}_{must} & = Damage_{must}(ACS^{avant_C}_{must})/ \\ \widetilde{ACS}^{après_C}_{must}(l, cb) & = & MIN(ACS^{sortie_C, entrée_C, \emptyset}_{must}(l, cb), ACS^{avant_C}_{must}[cb] \oplus DMG^{sortie_C}_{must}(l, cb)) \end{array}$$

$$\begin{array}{rcl} \forall l, cb & \widetilde{ACS}^{après_C}_{may} & = Damage_{may}(ACS^{avant_C}_{may})/ \\ \widetilde{ACS}^{après_C}_{may}(l, cb) & = & MIN(ACS^{sortie_C, entrée_C, \emptyset}_{may}(l, cb), ACS^{avant_C}_{may}[cb] \oplus DMG^{sortie_C}_{may}(l, cb)) \end{array}$$

$$\begin{array}{rcl} \forall l, cb & \widetilde{ACS}^{après_C}_{pers} & = Damage_{pers}(ACS^{avant_C}_{pers})/ \\ \widetilde{ACS}^{après_C}_{pers}(l, cb) & = & MAX(ACS^{sortie_C, entrée_C, \emptyset}_{pers}(l, cb), ACS^{avant_C}_{pers}[cb] \oplus DMG^{sortie_C}_{pers}(l, cb)) \end{array}$$

FIGURE 4.9: Dommage amélioré : étape intermédiaire

$$\forall l, cb\, DMG' = Update_{dmgOut}(DMG, B)\,/$$
$$DMG'(l, cb) = \begin{cases} \emptyset & si\, (line_B, block_B) = (l, cb) \vee DMG(l, cb) = \emptyset \\ DMG(l, cb) \oplus 1 & si\, line_B = l \wedge block_B \notin ACS^B_{must}(l) \\ DMG(l, cb) & sinon \end{cases}$$
$$\forall l, cb\, DMG' = Join_{dmgOut}(DMG1, DMG2)\,/$$
$$DMG'_{out}(l, cb) = MAX_N(DMG2_{out}(l, cb), DMG2_{out}(l, cb))$$
$$avec\, MAX_\emptyset = \lambda x \lambda y. \begin{cases} y & si\, x = \emptyset \\ x & si\, y = \emptyset \\ MAX(x, y) & sinon \end{cases}$$

FIGURE 4.10: Vieillissement de cache *out* (*must*)

$$\forall l, a' \in [0..A[\, \widetilde{ACS}^{après_C}_{must} = Damage_{must}(ACS^{avant_C}_{must})\,/$$
$$\widetilde{ACS}^{après_C}_{must}(l, a') = \{cb/a' = MAX_\emptyset(a \oplus DMG^{sortie_C}_{out}(l, cb), DMG^{sortie_C}_{in}(l, cb)\}$$

$$\forall l, a' \in [0..A[\, \widetilde{ACS}^{après_C}_{pers} = Damage_{pers}(ACS^{avant_C}_{pers})\,/$$
$$\widetilde{ACS}^{après_C}_{pers}(l, a') = \{cb/a' = MAX_\emptyset(a \oplus DMG^{sortie_C}_{out}(l, cb), DMG^{sortie_C}_{in}(l, cb)\}$$

$$avec\, \emptyset \oplus x = \emptyset\, \forall x$$

FIGURE 4.11: Dommage amélioré : étape finale

4.2.4.1 Fonction résumé pour le must, may, et *persistence* non-paramétrique

Afin de prendre en compte le contexte d'appel lors de la catégorisation des l-blocs du composant, nous avons introduit le concept de fonction *résumé*. Cette fonction a tout d'abord été introduite dans [6] pour les caches d'instructions à application directe, puis a ensuite été étendue aux caches associatifs par ensemble dans [7].

Comme indiqué précédemment, la fonction *résumé* peut être considérée comme une collection de catégories conditionnelles (CC). Une CC est associée à chaque l-bloc du composant et donne la catégorie finale du l-bloc en fonction du contexte (représenté par les ACS).

La catégorie conditionnelle a pour type $ACS_{pers} \times ACS_{must} \times ACS_{may} \rightarrow AH|AM|PS|NC$, les trois ACS fournis représentant l'état du cache lors de l'appel au composant. Pour définir cette fonction, introduisons d'abord une fonction appelée CAT, fonction de catégorisation. Cette fonction CAT est de type $ACS_{pers} \times ACS_{must} \times ACS_{may} \rightarrow AH \,|\, AM \,|\, PS \,|\, NC$, et elle prend en paramètre les ACS avant un l-bloc, pour donner en sortie la catégorie de celui-ci, de la même manière que celle qui est spécifiée dans la table 2.1 (*must* et *may* classique, et *persistence* non paramétrique).

De plus, soient $dommagePartiel_{may}$, $dommagePartiel_{must}$, et $dommagePartiel_{pers}$, fonctions respectivement de type $DMG_{may} \times ACS_{may} \times ACS_{may} \rightarrow ACS_{may}$, $DMG_{in} \times DMG_{out} \times ACS_{must} \rightarrow ACS_{must}$, et $DMG_{in} \times DMG_{out} \times ACS_{pers} \rightarrow ACS_{pers}$. Ces fonctions prennent en paramètre d'entrée les informations liées au dommage de cache à un point du composant donné (nommé p) ainsi que l'ACS à l'entrée du composant. Ces fonctions retournent l'ACS au point p. Ces fonctions sont une généralisation des fonctions $dommage_{may/must/pers}$, qui renvoient l'ACS à la sortie du composant, alors que $dommagePartiel_{may/must/pers}$ permet de les obtenir à n'importe quel point. Les fonctions $dommagePartiel_{may/must/pers}$ peuvent être implémentées de manière similaire à *dommage* (la seule différence est que les informations de dommage de cache utilisées sont celles correspondant au point p et fournies en paramètre, et non pas celles concernant le point de sortie du composant).

Ainsi, si pour un l-bloc lb du composant, on connaît les informations de dommage ($DMG^{lb}_{in} \times DMG^{lb}_{out} \times DMG^{lb}_{may} ACS^{lb}_{may}$) on peut exprimer la catégorie conditionnelle du l-bloc lb en fonction de ces informations sous la forme suivante :

$CC^{lb} = \lambda ACS^{entrée_C}_{pers} \lambda ACS^{entrée_C}_{must} \lambda ACS^{entrée_C}_{may} . CAT(ACS^{lb}_{may}, ACS^{lb}_{must}, ACS^{lb}_{pers})$, avec les valeurs ACS^{lb}_{may}, ACS^{lb}_{must}, et ACS^{lb}_{pers} calculées par les fonctions *dommagePartiel* (versions *may*, *must*, et *pers*). Ceci fonctionne mais n'est pas très performant, en effet il est nécessaire de créer une fonction CC^{lb} pour chaque l-bloc, et ensuite d'appliquer cette fonction pour chaque l-bloc lorsque le contexte d'appel est connu. Ceci n'est pas nécessaire : dans certains cas, la catégorie du l-bloc est indépendante du contexte d'entrée, soit parce que les fonctions *dommagePartiel*() indiquent que les ACS avant lb sont les mêmes quelles que soient les ACS à l'entrée du composant, soit parce que les diverses ACS possibles avant lb donnent la même catégorie une fois appliquées à la fonction CAT. Ceci se produit par exemple lorsque $block_{lb} \in ACS^{lb,entrée_C,\top}_{must}$, on sait que la catégorie de

lb sera AH dans tous les contextes, puisque l'analyse lancée sur le composant en prenant $\top$ comme état d'entrée garantit que l'ACS avant lb sera valide dans tous les contextes possibles. Le cas similaire existe pour la catégorie AM si $block_{lb} \notin ACS_{may}^{lb,entrée_C,\top}$

Notre implémentation d'analyse partielle dans OTAWA détecte ces cas, et génère alors une catégorie statique simple pour le l-bloc considéré, au lieu de créer une fonction résumé.

4.2.4.2 Cas de la *persistence* paramétrique

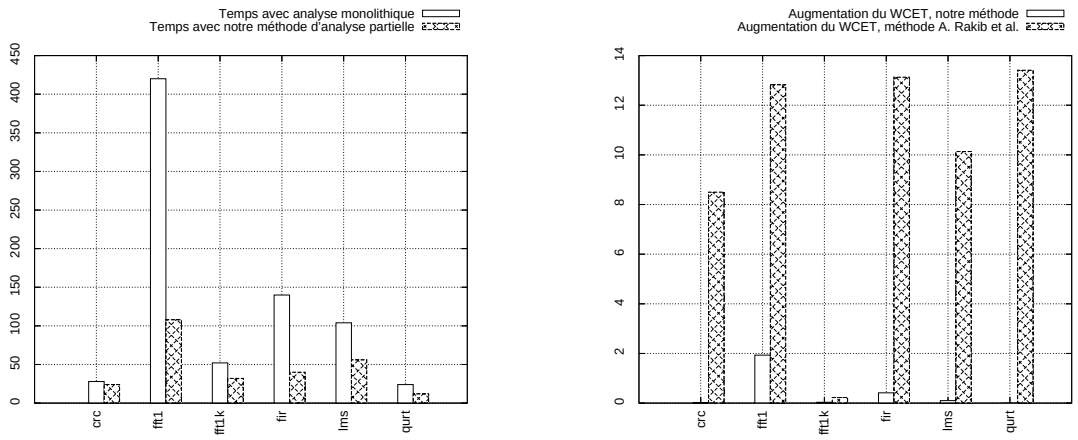
La prise en compte de la *persistence* paramétrique nécessite tout d'abord de réaliser l'analyse de *persistence* paramétrique sur le composant C, sans tenir compte du contexte (en prenant en entrée une ACS de *persistence* paramétrique ne contenant aucun élément), et donc en ignorant toute boucle qui pourrait contenir l'appel au composant. Ceci est réalisé lors de la phase d'analyse partielle, et l'information calculée donc stockée dans le résultat partiel associé au composant.

Lors de la composition, pour un l-bloc lb appartenant au composant, la fonction *résumé* devra calculer la catégorie de lb en fonction des ACS d'entrée : la fonction *résumé* est enrichie de telle manière que si cette catégorie résultante n'est pas AM ou AH, alors on réalise un traitement spécifique concernant la *persistence*. Deux cas mutuellement exclusifs se présentent :

1. Le l-bloc lb est dans (au moins) une boucle du composant, et la boucle la plus externe du composant contenant lb, notée L_c, est telle que lb n'est pas *persistent* par rapport à L_c (c'est-à-dire que $ACS_{multi}^{lb}[L_c] = l_\top$). Puisque lb n'est pas *persistent* par rapport à L_c, alors il ne sera pas *persistent* par rapport aux boucles contenant l'appel au composant. La catégorie de lb ne dépend pas du contexte, et est déterminée grâce ACS_{multi}^{lb}, en utilisant la règle de calcul de la *persistence* paramétrique décrite dans la section 3.1.3.2.
2. Le l-bloc lb n'est pas dans une boucle dans le composant, ou bien la boucle la plus externe du composant contenant lb, notée L_c, est telle que lb est *persistent* par rapport à L_{0c} (c'est-à-dire que $cache_{lb} \in ACS_{multi}^{lb}[L_c]$ et $ACS_{multi}^{lb}[L_c] \neq l_\top$). Puisque lb est *persistent* par rapport à L_c (ou bien que lb n'est dans aucune boucle au niveau du composant), il y a une chance que lb soit *persistent* par rapport à une boucle contenant l'appel au composant. L'$ACS_{multi}^{avant_C}$ contient n éléments, correspondant aux n boucles de L_0 à L_{n-1} qui contiennent l'appel au composant. Pour chaque boucle L_i, on a $ACS_{multi}^{lb}[L_i] = dommagePartiel_{pers}(ACS_{multi}^{entrée_C}[L_i], DMG_{in}^{lb}, DMG_{out}^{lb})$. Une fois obtenus les n éléments de $ACS_{multi}^{lb}[L_0]$ à $ACS_{multi}^{lb}[L_{n-1}]$, il est possible d'utiliser la règle décrite dans la section 3.1.3.2 pour obtenir la catégorie finale de lb.

4.2.5 Expérimentation

L'expérimentation que nous avons réalisé avec OTAWA a plusieurs buts. Tout d'abord, nous devons déterminer l'impact de l'analyse partielle sur le WCET obtenu (vérifier que le WCET calculé avec analyse partielle n'est pas plus faible que le WCET obtenu par



(a) Mesures du temps d'analyse

(b) Mesures du pessimisme

FIGURE 4.12: Expérimentation de l'analyse partielle du cache d'instructions

analyse monolithique, et estimer le pessimisme rajouté par l'analyse partielle). De plus, nous devons évaluer le gain de temps dû à l'analyse partielle. Nous rappelons que le gain de temps est dû à deux éléments : (1) le fait que les analyses ont généralement un temps d'exécution croissant de manière non linéaire, et (2) le fait que l'analyse partielle permet la réutilisation de résultats. Enfin, par souci d'exhaustivité, nous comparerons l'efficacité (en terme de précision du WCET) de notre fonction de dommage amélioré par rapport à celle proposé par A. Rakib et al. dans l'article [74]. Les mesures ont été réalisées avec la même architecture cible que dans l'expérimentation de la section 3.1.4.

La figure 4.12(a) compare le temps de calcul entre les deux approches, nous pouvons observer que le gain de temps est important surtout quand la fonction composant est appelée plusieurs fois. Par exemple, dans le cas de *fft1* le composant est une assez grosse fonction nommée *fft1()* qui est appelée deux fois et qui représente la plus grande partie du programme. Le composant de *fir* est une fonction *sin()* qui est appelée deux fois, et de plus ces appels sont dans les boucles. Par conséquent, lors de l'analyse de la boucle contenant l'appel, la fonction de dommage est utilisée, au lieu d'avoir à analyser le composant plusieurs fois jusqu'à la convergence vers le point fixe, ce qui accélère beaucoup l'analyse. Ces cas sont particulièrement adaptés à l'analyse partielle. En moyenne, l'analyse partielle permet de réaliser le calcul 2.33 fois plus vite.

La figure 4.12(b) montre l'augmentation du WCET due à l'analyse partielle (par rapport au WCET donné par analyse monolithique), et ce dans le cas de l'approche de A. Rakib et al. ainsi que dans le cas de notre approche. Ceci permet d'estimer le pessimisme [2] de chaque méthode. Pour calculer le WCET via l'analyse monolithique ainsi que via notre méthode d'analyse partielle, nous avons utilisé notre *persistence* paramétrique pour gérer

2. Ceci n'est qu'une estimation, car pour calculer le pessimisme réel, il aurait fallu calculer le pourcentage d'augmentation à partir du WCET réel, alors que nous l'avons calculé à partir du WCET obtenu par analyse monolithique, qui est une approximation.

les *First Miss*. Dans le cas de l'analyse de A. Rakib et al., par souci d'équité nous avons utilisé le déroulage de boucles (pour contrebalancer l'effet de l'absence de la *persistence*). Nous observons que le WCET obtenu par analyse partielle n'est jamais inférieur au WCET obtenu par analyse monolithique (ce qui serait inquiétant en ce qui concerne la validité des résultats), et que l'augmentation de pessimisme est faible. Un peu de pessimisme existe quand même (par rapport à l'analyse monolithique), car la fonction de dommage de cache n'est qu'une approximation : on a $ACS_{type}^{après_C} \subseteq damage(ACS_{type}^{avant_C})$ mais en général on n'a pas forcément $ACS_{type}^{après_C} = damage(ACS_{type}^{avant_C})$. Ceci est dû (entre autres) au fait que la fonction de dommage de cache calcule le vieillissement pour chaque bloc sans tenir compte de l'état du cache avant le composant : en réalité, s'il y a un accès à un bloc *cb* et qu'il existe d'autre blocs présents dans le même ensemble de cache à un âge plus vieux que *cb*, ces derniers blocs ne sont pas vieillis.

En moyenne, le WCET obtenu avec l'analyse partielle est de 0.42% supérieur au WCET obtenu par analyse monolithique. L'augmentation de WCET est plus importante avec la méthode de A. Rakib et al., nous constatons donc que notre fonction de dommage est plus avantageuse au niveau de la précision (0.42% d'augmentation du WCET contre 9.7% avec la méthode de A. Rakib et al.), tout en conservant la même rapidité de calcul.

Dans ce chapitre, nous avons présenté une méthode complète pour l'analyse partielle et la composition de la prédiction du comportement du cache d'instructions, avec les caches associatifs par ensembles. Cette approche présente deux avantages : (1) le temps d'analyse est fortement réduit par rapport à une analyse monolithique et (2) ceci est un premier pas vers le support des COTS - Components Off The Shelf - dans l'analyse de WCET. De plus, on constate que le pessimisme n'est que très peu augmenté par l'analyse partielle.

Chapitre 5

Autres analyses partielles

Le chapitre précédent a présenté l'analyse partielle du comportement du cache. Ce chapitre présente les analyses partielles restantes, comme l'analyse des limites de boucles et du prédicteur de branchement.

5.1 Limites de boucles

Dans cette section nous présentons l'analyse partielle des limites de boucles. Tout d'abord, nous rappelons la méthode d'analyse des limites de boucle par interprétation abstraite présentée dans la section 2.3. Ensuite nous présentons notre méthode d'analyse partielle, et puis nous décrivons l'expérimentation réalisée, ainsi que les résultats.

5.1.1 Contexte et définition du problème

Comme vu dans la section 2.3.1, l'interprétation abstraite utilisée pour réaliser l'analyse des bornes de boucle est basée sur un domaine abstrait appelé *AbstractStore* qui associe à chaque identificateur une expression symbolique représentant sa valeur.

L'analyse est réalisée en trois étapes :

- la construction de l'arbre des contextes, la détection de l'incrément et des variables d'induction de chaque boucle, et la construction de l'expression initiale des bornes
- la construction des expressions finales du *max* et du *total* de chaque boucle, avec prise en compte du contexte
- l'évaluation numérique des expressions du *max* et du *total* pour chaque boucle.

5.1.2 Notre approche pour les limites de boucles

Pour rendre l'évaluation des limites de boucles adaptables à l'analyse partielle de composants, il faut :

- Pouvoir déterminer l'influence de l'appel au composant sur l'état du programme (effets de bords, variables globales modifiées, paramètres passés par référence), car ces effets de bords peuvent avoir une influence sur le calcul de l'incrément dans l'étape 1, ou bien sur l'interprétation abstraite réalisée dans l'étape 2.

```
extern int x,y;
int myComp() {
int z,t;
z = 3;
t = z - 2;
x = x + z;
for (int i = 0; i < x; x++) {
...
}
return y + t;
}
```

FIGURE 5.1: Exemple : composant et bornes de boucle

- Fournir des expressions de bornes de boucles du composant en fonction du contexte d'appel du composant. Grâce à ces expressions paramétrées, il devient possible d'instancier les bornes *max* et *total* des boucles du composant une fois connu le contexte d'appel, afin d'obtenir une valeur numérique.

Le premier point sera traité par la fonction *transfert*, tandis que le second sera pris en compte par la fonction *résumé*.

5.1.2.1 Fonction transfert

Pour gérer les effets de l'appel du composant sur le programme principal, la fonction *transfert* de l'analyse des bornes de boucles doit prendre en paramètre l'*AbstractStore* avant l'appel au composant (le contexte), et retourner l'*AbstractStore* après l'appel (l'expression des variables après l'appel au composant, pour le contexte donné). Pour ce faire, rappelons que les *AbstractStore* sont composables. L'opérateur de composition, noté $\circ$, permet à partir d'un *AbstractStore AS,* d'écrire $AS\prime = ctx \circ AS$, le résultat $AS\prime$ sera l'*AbstractStore AS* évalué dans l'*AbstractStore* du contexte ctx.

Par conséquent, si la construction de l'AS d'un composant *comp* donne AS_{comp} (par la méthode d'interprétation abstraite donnée dans la section 2.3.1), alors pour tout identificateur id qui est défini par AS_{comp}, $AS_{comp}[id]$ donne l'expression de celui-ci en fonction du contexte, et $(ctx \circ AS_{comp})[id]$ permet d'avoir l'expression de id une fois le contexte appliqué. Ceci implique donc que la fonction $\lambda ctx.(ctx \circ AS_{comp})$ est bien la fonction *transfert* recherchée, pour le composant *comp* : elle prend un contexte, et renvoie le contexte modifié par le passage à travers *comp*.

Toutefois, AS_{comp} contient l'expression de toutes les variables modifiées par le composant, y compris celles dont la portée ne sort pas du composant. Ceci alourdit inutilement la fonction de transfert. On peut, à partir d'AS_{comp}, éliminer les variables qui ne sont pas susceptibles d'avoir un effet sur le programme principal (pour ne garder que le strict minimum), et stocker ce qui reste dans le résultat partiel du composant. Les variables pouvant avoir un effet sont les variables globales et statiques, les variables dont l'adresse est contenue dans un pointeur, les variables statiques, la valeur de retour de la fonction point d'entrée du composant.

Par exemple, soit la fonction (composant) montrée dans la figure 5.1. L'*AbstractStore* de ce composant sera $[z \to 3; t \to z-2; x \to x+z; RES_{comp} \to y+t]$. On élimine ensuite les variables locales (z et t) et on exprime le reste des variables en fonction de z et t, on a donc $transfert_{comp}(ctx)$=$ctx \circ [x \to x+3; RES_{comp} \to y+1]$. Prenons par exemple un contexte d'entrée spécifié par l'*AbstractStore* $[x \to 10; y \to 20]$. L'*AbstractStore* de sortie sera $transfert_{comp}([x \to 10; y \to 20])$=$[x \to 10; y \to 20] \circ [x \to x+3; RES_{comp} \to y+1]$=$[x \to 13; y \to 20; RES_{comp} \to 21]$, ce qui correspond bien à ce que l'on recherche : la composabilité des *AbstractStore* permet d'obtenir facilement l'effet de l'appel du composant sur le reste du programme.

5.1.2.2 Fonction résumé

La fonction *résumé* est nécessaire pour obtenir les bornes de boucles en fonction du contexte d'appel. Compte tenu de l'algorithme utilisé pour l'analyse, ceci est facile : en effet, dans la section 2.3.1, on voit que le *max* et le *total* définitif de chaque boucle est calculé lors de l'étape 2. C'est l'étape 3 qui réalise le calcul numérique. Donc, si on interrompt l'analyse entre l'étape 2 et 3, on obtient les expressions non évaluées du *max* et du *total*. Ces expressions peuvent contenir des variables libres, c'est-à-dire des variables correspondant au contexte (variables globales externes, paramètres d'entrée de la fonction, ...).

Par exemple, si on réalise l'analyse jusqu'à l'étape 2 incluse, la boucle présente dans cette fonction aura pour *max* (et pour *total* aussi) l'expression $x+3$, x représentant ici la valeur de x avant l'appel à la fonction. L'expression $x+3$ représente bien l'expression du *max* et du *total* de la boucle en fonction du contexte.

Le résultat de l'analyse des bornes de boucles peut être vu comme une fonction qui associe à chaque identifiant i (identifiant de la boucle L_i), une paire *(max,total)* : $resultatAnalyse(i) = (max_i, total_i)$. La fonction *résumé* devrait donc prendre un *AbstractStore* contexte en entrée, et renvoyer le résultat d'analyse du composant, autrement dit, être de type $AS \times IDBOUCLE \to ENTIER \times ENTIER$. Notons $resComp$ le résultat de l'analyse du composant stoppé à l'étape 2 (fonction prenant en entrée un ID de boucle, et renvoyant les expressions finales du *max* et du *total* de cette boucle). La fonction *résumé* peut alors être définie comme suit : $résumé(ctx, i) = ctx[resComp(i)]$.

5.1.2.3 Composition

Les sections précédentes ont permis de définir la fonction *transfert* et la fonction *résumé*, qui constituent le résultat partiel associé à un composant, dans le cadre de l'analyse des limites de boucles.

Considérons maintenant la situation dans laquelle on possède un programme faisant appel à un composant dont on connaît le résultat partiel, sans avoir accès à son code. Étudions d'abord la prise en compte de la fonction *transfert* dans la composition, l'utilisation de la fonction *résumé* sera traitée plus tard.

La fonction *transfert* du composant intervient, lors de la composition, pendant les étapes 1 et 2 de l'analyse du programme principal. Lors de l'étape 1, une analyse par interprétation abstraite sur les *AbstractStore* est effectuée pour chaque boucle, afin de trouver l'incrément des variables d'induction. Si une boucle contient un appel au composant, alors le code du composant ne sera pas disponible lors de cette interprétation abstraite. On modélise donc l'appel au composant par une simple instruction (virtuelle) dont la fonction *Update* est donnée par $Update(AS) = transfert_{comp}(AS)$. Ceci permet à l'interprétation abstraite de fonctionner malgré tout, et ainsi d'obtenir les incréments des variables d'induction. Lors de l'étape 2, l'interprétation abstraite sur les *AbstractStore* est à nouveau utilisée pour calculer les expressions définitives des *max* et *total* de chaque boucle. On modélise également les appels à un composant par une instruction dont la fonction *Update* est donnée par la fonction de transfert.

Tout ceci permet d'obtenir les expressions du *max* et du *total* pour les boucles du programme principal, en tenant compte des éventuels effets de bords causés par le composant. Toutefois, il reste à fournir les expressions des bornes de boucles du composant lui-même. C'est là qu'intervient la fonction *résumé*. Rappelons que lors de l'étape 2 de l'analyse du programme principal, on réalise un parcours des boucles de la plus interne jusqu'à la plus externe et ce pour chaque nid. Chaque boucle L_i est annotée par les bornes temporaires $maxB_i$ et $totalB_i$, et ces expressions sont placées dans le contexte de boucles de plus en plus externe au fur et a mesure du parcours. Maintenant, supposons que lors de ce parcours, on arrive à une boucle L_e qui contient un appel au composant. Les éventuelles boucles internes de L_e qui ne sont pas dans le composant ont été ramenées au contexte de la boucle L_e par l'algorithme traditionnel de l'étape 2. Grâce à la fonction résumé on a les expressions $maxB_{ci}$ et $totalB_{ci}$ (paramétrées par le contexte d'appel du composant) pour chaque boucle L_{ci} du composant. Ces expressions sont modifiées pour les ramener au contexte de la boucle L_e, de manière similaire au traitement des boucles internes : soit $varB_e$ le compteur d'itération de la boucle L_e, pour chaque $maxB_{ci}$ et $totalB_{ci}$, ces expressions deviennent respectivement $MAX(varB_e = 0..expBorne_e, maxB_{ci})$ et $\sum_{varB=0}^{expBorne_e} totalB_{ci}$ (où $expBorne_e$ est l'expression de la borne de boucle L_e normalisée qui a été calculée lors de l'étape 1). Après cette opération, nous disposons de l'expression des bornes des boucles internes de L_e (boucles du composant ainsi que boucles hors composant), et il est possible de réaliser l'interprétation abstraite de L_e comme présenté dans la section 2.3.1. Lors de cette analyse, la fonction *Update* des boucles internes sera définie par la composition avec l'*AbstractStore* stockée dans l'annotation de la boucle interne, et l'*Update* correspondant à un appel de composant sera défini par la fonction *transfert* du composant.

5.1.3 Expérimentation

Nous avons expérimenté l'analyse partielle des bornes de boucles avec oRange sur les benchmarks Mälardalen qui possédaient une taille suffisante pour fournir des composants intéressants contenant des boucles : *adpcm, cnt, crc, duff, expint, jfdctint,* ainsi

que *ludcmp*. Sur tous ces tests, les estimations de bornes de boucles ont donné les mêmes résultats avec et sans analyse partielle (aussi bien pour le *max* que pour le *total*), c'est pourquoi nous ne donnerons pas de tableau comparatif ici. Ceci montre que, pour les benchmarks considérés, l'analyse partielle des bornes de boucles n'ajoutera pas de pessimisme. De plus, sur les quelques tests ayant une taille suffisamment importante (*adpcm, compress*), on observe qu'en moyenne l'estimation des bornes de boucles par analyse partielle est 1.5 fois plus rapide (en tenant compte du temps d'analyse partielle et du temps de composition). Les autres tests sont trop petits et sont trop rapidement traités par oRange pour pouvoir mesurer une augmentation des performances de manière fiable.

5.2 Prédiction de branchements

Dans cette partie, nous présentons notre méthode pour la prise en compte de la prédiction de branchement, qui est une adaptation à IPET de la méthode présentée par I. Puaut et al. dans [15] et rappelée dans la section 2.6. Ensuite nous présentons l'adaptation de cette méthode à l'analyse partielle, puis nous montrons des résultats expérimentaux.

5.2.1 Prédiction de branchement et IPET

L'analyse de prédiction de branchement présentée par I. Puaut et al. dans [15] n'est pas conçue pour la méthode IPET. Toutefois, notre outil de calcul de WCET (OTAWA) favorise principalement IPET (bien que ce ne soit pas la seule méthode qu'il supporte) et, de plus, le principe général d'analyse partielle que nous avons développé est davantage adaptée à IPET qu'aux autres méthodes de calcul.

C'est pourquoi, avant de nous attaquer à l'étude de l'analyse partielle de la prédiction de branchement, nous présentons notre méthode d'analyse de la prédiction de branchement, qui est en fait une adaptation de la méthode présentée par I. Puaut et al. dans [15], de manière à la rendre compatible avec la méthode IPET.

Cette adaptation se comporte de manière similaire à l'analyse du cache d'instructions : dans un premier temps, nous construisons des ABS (similaires à ceux de I. Puaut et al.) pour savoir quelles instructions de branchement peuvent se trouver dans la BHT à chaque point du programme, ensuite ces ABS sont utilisées pour classer chaque instruction de branchement dans une catégorie. Finalement, ces catégories sont utilisées pour rajouter des contraintes dans le système ILP, afin de prendre en compte les pénalités dues aux prédictions de branchement incorrectes. Dans toute la description de cette analyse, quand nous utiliserons le terme « instruction de branchement », nous désignerons les instructions de branchement conditionnelles et directes. En effet, les instructions de branchement qui sont à la fois inconditionnelles et directes, n'ont pas besoin de prédiction. Les instructions de branchement indirectes ne sont pas supportées par cette analyse.

5.2.1.1 Construction des ABS

Comme dans la méthode de I. Puaut et al., les ABS sont calculés avant et après chaque BB. Comme pour l'analyse du cache d'instructions par interprétation abstraite, trois analyses distinctes sont réalisées (*may*, *must*, et *pers*). Nous utilisons la même notation que pour le cache pour désigner le contenu des ABS, $ABS^p_{type}(l)$ où l est la ligne dans la BHT, p le point du programme (ou de manière équivalente, l'instruction de branchement concernée par l'ABS). Les ABS *may*, *must*, et *pers* ont la même signification que pour le cache d'instructions : chaque ABS associe un ensemble s d'instructions de branchement à chaque ligne dans la BHT, un âge $a \in [0..A[$ est également associé à chaque instruction (A représente ici le niveau d'associativité de la BHT) pour le *may* et le *must*, le *pers* faisant intervenir un âge $l_\top$ supplémentaire. Comme pour le cache d'instructions, dans le *may*, $ABS^p_{may}(l)$ contient toutes les instructions qui peuvent être dans la ligne l, et l'âge associé est une borne inférieure de l'âge réel. Dans le cas du *must*, $ABS^p_{must}(l)$ contient les instructions qui doivent être dans la ligne l, et l'âge associé représente une borne supérieure. Dans le cas du *pers*, $ABS^p_{pers}(l)$ contient les instructions qui peuvent être dans la BHT à la ligne l (comme pour le *may*), mais l'âge associé est une borne supérieure de l'âge réel (comme pour le *must*). L'âge virtuel $l_\top$ qui est rajouté pour le *pers* a la même signification que pour le cache : il signifie qu'une instruction peut avoir été chargée dans la BHT, puis ensuite vieillie jusqu'à ce qu'elle en sorte.

Des fonctions *Update* et *Join* distinctes existent pour chaque analyse. Les fonctions *Update* prennent en entrée un ABS ainsi qu'un bloc de base (il ne peut y avoir, au maximum, qu'une instruction de branchement par bloc de base), et retournent l'ABS modifiée par le chargement de l'instruction de branchement. La fonction *Update* se charge d'insérer la nouvelle instruction dans l'ABS, et de vieillir, le cas échéant, les autres instructions qui étaient présentes dans la même ligne. Une description précise des fonctions *Update* pour le *may*, *must*, et *pers* de l'analyse de prédiction de branchement est fournie dans la figure 5.2.

La fonction *Join* prend en entrée deux ABS, et renvoie une ABS en sortie. Le *Join* du *may* réalise l'union des ensembles ligne par ligne, en prenant l'âge le plus jeune lorsque la même instruction se retrouve dans les deux ABS d'entrée. Le *Join* du *must* réalise l'intersection, et prend l'âge le plus vieux. Le *Join* du *pers*, réalise l'union, et prend l'âge le plus vieux (en considérant $l_\top$) comme le plus vieux des âges. La définition exacte du *Join* pour les analyses *may*, *must*, et *pers* est donnée dans la figure 5.3.

Pour l'instant, la définition de l'analyse *pers* que nous avons donnée n'est pas paramétrique, c'est une *persistence* externe. C'est pourquoi, afin d'augmenter la précision de l'analyse, nous avons choisi d'utiliser la *persistence* paramétrique également pour l'analyse de prédiction de branchement. Convertir la *persistence* de l'analyse de prédiction de branchement en *persistence* paramétrique se fait de manière identique à ce qui est fait pour le cache d'instruction, section 3.1.3.2 : l'ABS *pers* devient une pile d'ABS de *persistence* externe, et il y a empilement / dépilement des éléments lorsqu'on entre dans une boucle ou qu'on la quitte.

$Update_{must}(ABS, instr)(l, a) = e(a)$, *tel que* :

$$
\begin{array}{rcl}
si & \exists h/ & instr \in e(h) : \\
e(0) & = & \{instr\}, \\
e(i) & = & ABS(l, i-1)/i = [1..h[\\
e(h) & = & ABS(l, h-1) \cup ABS(l, h) - instr \\
e(i) & = & ABS(l, i)/i \in]h..A[\\
si & \forall h & instr \notin e(h) : \\
e(0) & = & \{instr\}, \\
e(i) & = & ABS(l, i-1)/i \in [1..A[
\end{array}
$$

$Update_{may}(ABS, instr)(l, a) = e(a)$, *tel que* :

$$
\begin{array}{rcl}
si & \exists h/ & instr \in e(h) : \\
e(0) & = & \{instr\}, \\
e(i) & = & ABS(l, i-1)/i = [1..h] \\
e(h+1) & = & ABS(l, h+1) \cup ABS(l, h) - instr \\
e(i) & = & ABS(l, i)/i \in]h+1..A[\\
si & \forall h & instr \notin e(h) : \\
e(0) & = & \{instr\}, \\
e(i) & = & ABS(l, i-1)/i \in [1..A[
\end{array}
$$

$Update_{pers}(ABS, instr)(l, a) = e(a)$, *tel que* :

$$
\begin{array}{rcl}
si & \exists h/ & instr \in ABS^{instr}_{must}(l, h) : \\
e(0) & = & \{instr\}, \\
e(i) & = & ABS(l, i-1)/i = [1..h[\\
e(h) & = & ABS(l, h-1) \cup ABS(l, h) - instr \\
e(i) & = & ABS(l, i)/i \in]h..A[\\
si & \forall h & instr \notin ABS^{instr}_{must}(l, h) : \\
e(0) & = & \{instr\}, \\
e(i) & = & ABS(l, i-1)/i \in [1..A[
\end{array}
$$

FIGURE 5.2: *Update* de l'analyse de prédiction de branchements

$$\forall l, a\ Join_{must}(ABS_1, ABS_2)(l, a) =$$
$$\{instr : \exists a_1, a_2/instr \in ABS_1(l, a_1),\ instr \in ABS_2(l, a_2),\ a = max(a_1, a_2)$$

$$\forall l, a\ Join_{may}(ABS_1, ABS_2)(l, a) =$$
$$\begin{aligned}\{instr : \exists a_1, a_2/instr &\in ABS_1(l, a_1),\ instr \in ABS_2(l, a_2),\ a = min(a_1, a_2)\} \cup \\ \{instr : instr &\in ABS_1(l, a),\ \forall a'\ instr \notin ABS_2(l, a')\} \cup \\ \{instr : instr &\in ABS_2(l, a),\ \forall a'\ instr \notin ABS_1(l, a')\}\end{aligned}$$

$$\forall l, a\ Join_{pers}(ABS_1, ABS_2)(l, a) =$$
$$\begin{aligned}\{instr : \exists a_1, a_2/instr &\in ABS_1(l, a_1),\ instr \in ABS_2(l, a_2),\ a = max(a_1, a_2)\} \cup \\ \{instr : instr &\in ABS_1(l, a),\ \forall a'\ instr \notin ABS_2(l, a')\} \cup \\ \{instr : instr &\in ABS_2(l, a),\ \forall a'\ instr \notin ABS_1(l, a')\}\end{aligned}$$

FIGURE 5.3: *Join* de l'analyse de prédiction de branchements

5.2.1.2 Catégorisation

Les ABS ainsi calculées sont utilisées pour classer chaque instruction de branchement (appelée *instr* dans cet exemple, et se situant dans le bloc de base *bb*) dans une des catégories suivantes :

- *Always D-Predicted* (ADP) est utilisé lorsque l'interprétation abstraite indique que l'instruction ne sera jamais dans la BHT lors de son exécution. Ceci se produit si $instr \notin ABS^{bb}_{may}(ligne_{instr})$.
- *First Unknown* (FU) est utilisé lorsque la première fois qu'une instruction de branchement est exécutée dans une boucle, on ne sait pas si elle sera dans la BHT, mais par contre on est sûr qu'elle sera *H-Predicted* pour toutes les autres itérations de la boucle. Cette catégorie est similaire à la catégorie *Persistent* du cache, et est aussi associée à une tête de boucle. Pour attribuer cette catégorie à une instruction, il faut d'une part que les conditions pour la catégorie *Always D-Predicted* ne soient pas satisfaites. De plus, définissons une fonction $IsPers()$, tel que $IsPers(instr, L_i) = \exists a/instr \in ABS^{BB}_{pers}[L_i](l, a) \wedge a \neq l_\top$. S'il existe une boucle L_i tel que $IsPers(instr, L_i) \wedge \forall L_j,\ L_i \subset L_j \rightarrow \neg IsPers(instr, L_j)$, alors l'instruction est catégorisée *First Unknown*(L_i).
- *Not Classified* (NC) est utilisé quand aucun autre cas ne s'applique, et signifie qu'on n'a pas d'information sur la présence (ou non) de l'instruction de branchement dans la BHT.

On remarque que l'ABS *must* n'est pas utilisé lors de la catégorisation, toutefois il doit quand même être calculé, car le résultat du *must* est utilisé pour calculer les ABS de *persistence*.

5.2.1.3 Génération des contraintes ILP

Pour adapter l'analyse de prédiction de branchement à la méthode IPET, tout d'abord on crée une variable $x^i_{misspred}$ pour chaque instruction de branchement i, correspondant au bloc de base numéro i (nous rappelons que cette correspondance est possible car il n'existe au plus qu'une instruction de branchement par bloc de base). Cette variable $x^i_{misspred}$ représente le nombre de mauvaises prédictions pour l'instruction de branchement i. Soit B l'ensemble des blocs de base se terminant par une instruction de branchement, la fonction objectif est enrichie des termes $\sum_{i \in B} t^i_{misspred} \times x^i_{misspred}$, où $t^i_{misspred}$ représente la pénalité en nombre de cycles d'une mauvaise prédiction pour l'instruction i.

Il faut aussi générer des contraintes ILP pour chaque instruction de branchement, selon sa catégorie. Dans tous les cas, la contrainte $x^i_{misspred} \leq x^i$ est générée, pour représenter le fait qu'une instruction de branchement ne peut pas causer plus de mauvaises prédictions que le nombre de fois qu'elle est exécutée. Les contraintes générées dépendent, en plus de la catégorie, du contexte dans lequel cette instruction de branchement est utilisée (boucle contenant l'instruction) . Ces contraintes serviront à borner la valeur de $x^i_{misspred}$. Pour chaque instruction *instr*, selon la catégorie un traitement différent (exposé dans les paragraphes ci dessous) est effectué.

5.2.1.4 La catégorie Always D-Predicted

Si l'instruction est classée dans cette catégorie, alors la prédiction sera toujours la même, c'est-à-dire la prédiction par défaut (qui est le plus souvent *non pris*). C'est pourquoi le nombre de mauvaises prédictions est égal au nombre de fois qu'on empruntera l'arc *pris* partant de x^i. Soit (i, j) cet arc *pris*, alors on génère une contrainte ILP $x^i_{misspred} = e_{i,j}$.

5.2.1.5 La catégorie First Unknown

Dans cette catégorie, le premier accès à l'instruction peut être prédit par l'historique ou bien prédit par défaut. Par contre, les prochains accès seront prédits par historique, et ce jusqu'à la prochaine entrée dans la boucle L_i (c'est-à-dire la prochaine itération de la boucle contenant L_i) associée à la catégorie *First Unknown* de cette instruction. Considérons l'ensemble IL des instructions du CFG ayant la même adresse qu'*instr* (la situation dans laquelle plusieurs instructions du CFG ont la même adresse peut se produire en cas d'*aliasing* de blocs de base, par exemple lors du déroulement de boucles ou de l'inlining de fonctions), ayant la même catégorie qu'*instr* (y compris la boucle associée au *First Unknown*). Le premier accès à une instruction de IL sera soit prédit par l'historique, soit prédit par défaut. Ensuite, toutes les instructions dans IL seront prédites par historique (jusqu'à la prochaine entrée dans L_i). En d'autres mots, il y aura pour chaque entrée dans L_i, au maximum une seule prédiction par défaut pour tout IL. Le cas de figure où $|LI| > 1$ est similaire au cas des blocs liés de l'analyse de cache (voir la première partie de la section 3.1.3.3).

Pour borner le nombre de mauvaises prédictions pour les instructions de IL, essayons

de déterminer quelles séquences d'exécutions d'arcs *pris* (P*)* et *non-pris* (NP) mènent au nombre maximum de mauvaises prédictions. On peut voir facilement que la situation la plus défavorable est celle où on alterne entre les deux états du prédicteur *(faiblement) pris* et *(faiblement) non-pris*. On ne connaît pas l'état initial du prédicteur, mais on voit que pour aboutir au nombre maximum de mauvaises prédictions, il faudrait partir soit de l'état *fortement pris* soit de l'état *fortement non pris.* Considérons que l'on part de l'état *fortement pris* (le cas *fortement non-pris* est similaire) il faut faire deux branchements *non-pris* jusqu'à arriver à l'état *non-pris*, et ensuite alterner entre *pris* et *non pris*, ce qui provoque une mauvaise prédiction à chaque branchement. Pour chaque cycle de *pris/non-pris* on a un branchement *pris*, et deux mauvaises prédictions. De plus, il faut compter les deux mauvaises prédictions liées aux deux branchements *non-pris* du début. Le nombre de mauvaises prédictions est donc égal à $2+2\times p$ (où p représente le nombre de branchements pris). Pour des raisons similaires, le nombre de mauvaises prédictions est borné par $2 + 2 \times np$ (où np représente le nombre de branchements non pris).

On peut donc générer les contraintes $\sum_{i\in IL} x^i_{misspred} \leq 2 \times \sum_{e\in entrées(L_i)} e + 2 \times \sum_{i\in IL} e_{i,np}$, où $entrées(L_i)$ est l'ensemble des arcs d'entrée de L_i, et où $e_{i,np}$ est la variable ILP associée à l'arc *non-pris* de l'instruction de branchement i. De manière similaire, on génère également la contrainte $\sum_{i\in IL} x^i_{misspred} \leq 2 \times \sum_{e\in entrées(L_i)} e + 2 \times \sum_{i\in IL} e_{i,p}$, où $e_{i,p}$ est la variable ILP associée à l'arc *pris* de l'instruction de branchement i.

5.2.1.6 La catégorie Not-Classified

Dans ce cas de figure, on ne sait rien sur cette instruction de branchement, on peut donc simplement dire que le nombre de mauvaises prédictions ne peut pas être supérieur au nombre de fois où l'instruction de branchement est exécutée, ce qui est déjà représenté par la contrainte ILP par défaut, $x^i_{misspred} \leq x^i$.

5.2.2 Prédiction de branchement et analyse partielle

Comme pour le cache, l'analyse partielle de la prédiction du branchement nécessite une fonction *transfert* pour savoir l'effet qu'a l'appel à un composant sur les tables du prédicteur. La fonction *résumé* permettra de savoir, en fonction des ABS à l'entrée du composant, les catégories des instructions de branchement de celui-ci. En ce qui concerne le problème de l'implantation du code du composant en mémoire, la solution apportée est la même que pour l'analyse partielle du cache : lors de l'analyse partielle, on considère que l'adresse d'implantation est 0, et ensuite lors de la composition, une simple translation des numéros d'ensembles permettront d'aboutir aux résultats corrects. Il n'y a pas de restriction sur l'implantation en mémoire du composant, car contrairement au cas du cache, dans la prédiction de branchement il n'y a pas de concept de *blocs de cache*, une instruction correspond toujours à un ensemble de BHT différent de l'instruction qui la précède (ou suit) .

5.2.2.1 Fonction transfert

Pour la fonction de transfert, nous utilisons la même approche que pour le cache, basée sur l'information de vieillissement (vieillissement des instructions de branchement dans les ensembles de la BHT) et sur les instructions insérées (instructions de branchement du composant se retrouvant dans la BHT à la sortie de celui-ci). Chaque fonction de transfert (*may*, *must*, et *pers*) prend en entrée un ABS, et renvoie un autre ABS en sortie. L'analyse de *persistence* que nous décrivons ici est la *persistence* externe, pour dériver l'analyse partielle de *persistence* paramétrique à partir de la *persistence* externe, nous procédons de la même manière que pour le cache (voir section 4.2.3.6).

Le dommage *may* nécessite le calcul préalable des instructions insérées, $ABS_{may}^{sortie_C, entrée_C, \emptyset}$, calculées en lançant une analyse *may* sur le composant en prenant comme état d'entrée l'ensemble vide. Le vieillissement du *may* associe à chaque instruction de branchement un vieillissement entier qui représente le minimum possible du vieillissement qui résulte de l'exécution du composant. Les fonctions *Update* et *Join* du vieillissement *may*, ainsi que la fonction de dommage résultante sont données dans la figure 5.4.

Le dommage *must* ne nécessite pas le calcul des instructions insérées, et ce pour les mêmes raisons que celles évoquées dans le chapitre sur l'analyse partielle du cache. Par contre, l'information de vieillissement est séparée en deux parties, le DMG_{in} et le DMG_{out}, prenant en compte respectivement les chemins qui passent, et qui ne passent pas, par l'instruction de branchement considérée par le DMG. Le DMG_{in} est tiré des ABS de *persistence*, et le DMG_{out} est calculé grâce à une interprétation abstraite dont les fonctions *Update* et *Join* sont données figure 5.5. A partir du DMG_{in} et du DMG_{out}, il est possible d'obtenir la fonction de dommage du *must* comme précisé dans la figure 5.6.

Pour le dommage *pers*, les instructions vieillissent dans les ABS de la même manière que pour le *must,* et donc la fonction de dommage peut être dérivée à partir du vieillissement du *must*, comme indiqué dans la figure 5.6. La signification des opérateurs MIN, MAX et $\oplus$ dans cette section est la même que leur signification dans la partie traitant de l'analyse partielle du cache d'instructions.

5.2.2.2 Fonction résumé

Comme pour le cache, la fonction *résumé* peut être vue comme une collection de Catégories Conditionnelles (une CC associée à chaque instruction de branchement du composant), ayant chacune pour type $ABS_{pers} \times ABS_{may} = \{ADP \,|\, FU \,|\, NC\}$ (le ABS_{must} n'est pas utilisé pour la catégorisation), prenant en paramètre les ABS à l'entrée du composant, et renvoyant la catégorie de l'instruction de branchement. Soit, la fonction CAT de type $ABS_{pers} \times ABS_{may} = \{ADP \,|\, FU \,|\, NC\}$ prenant en paramètre les ABS avant un bloc contenant une instruction de branchement, et renvoyant la catégorie de celle-ci, calculée par la méthode décrite dans la section 5.2.1.2. Soient les deux fonctions $dommagePartiel_{pers}$ et $dommagePartiel_{may}$, de type respectivement $DMG_{in} \times DMG_{out} \times ABS_{pers} \rightarrow ABS_{pers}$ et $DMG_{may} \times ABS_{may} \times ABS_{may} \rightarrow ABS_{may}$. Ces fonctions suivent le même principe que leurs versions pour le cache d'instructions, et renvoient l'ABS devant

$$\forall l, instr\, DMG' = Update_{may}(DMG, instr)/DMG'(l, instr\prime) =$$
$$\begin{cases} 0 & instr = instr\prime \\ DMG(l, instr\prime) & si\, ligne_{instr} \neq l \\ DMG(l, instr\prime) & si\, ligne_{instr} = l \wedge instr \in ABS^{instr}_{may}(l) \\ DMG(l, instr\prime) \oplus 1 & sinon \end{cases}$$
$$\forall l\, DMG' = Join_{may}(DMG1, DMG2)\,/$$
$$DMG'(l) = MIN(DMG1(l), DMG2(l))$$

$$\begin{array}{rcl} \forall l, instr \quad ABS^{après_C}_{may} & = & Damage_{may}(ABS^{avant_C}_{may})/ \\ \widetilde{ABS}^{après_C}_{may}(l, instr) & = & MIN(ABS^{sortie_C, entrée_C, \emptyset}_{may}(l, instr), ABS^{avant_C}_{may}[cb] \\ & & \oplus DMG^{sortie_C}_{may}(l, instr)) \end{array}$$

FIGURE 5.4: Vieillissement et dommage *may*

$$\forall l, instr\, DMG' = Update_{dmgOut}(DMG, instr)\,/$$
$$DMG'(l, instr\prime) = \begin{cases} N & si\, instr = instr\prime \vee DMG(l, instr) = N \\ DMG(l, instr\prime) \oplus 1 & si\, ligne_{instr} = l\ \wedge instr \notin ABS^{instr}_{must}(l) \\ DMG(l, instr\prime) & sinon \end{cases}$$
$$\forall l, instr\, DMG' = Join_{dmgOut}(DMG1, DMG2)\,/$$
$$DMG'_{out}(l, instr) = MAX_N(DMG2_{out}(l, instr), DMG2_{out}(l, instr))$$
$$avec\, MAX_\emptyset = \lambda x \lambda y. \begin{cases} y & si\, x = \emptyset \\ x & si\, y = \emptyset \\ MAX(x, y) & sinon \end{cases}$$

FIGURE 5.5: Vieillissement DMG_{out}

$$\forall l, a' \in [0..A[\, \widetilde{ABS}^{après_C}_{must} = Damage_{must}(ABS^{avant_C}_{must})\,/$$
$$\widetilde{ABS}^{après_C}_{must}(l, a') = \{instr/a' = MAX_\emptyset(a \oplus DMG^{sortie_C}_{out}(l, instr), DMG^{sortie_C}_{in}(l, instr)\}$$

$$\forall l, a' \in [0..A[\, \widetilde{ABS}^{après_C}_{pers} = Damage_{pers}(ABS^{avant_C}_{may})\,/$$
$$\widetilde{ABS}^{après_C}_{pers}(l, a') = \{instr/a' = MAX_\emptyset(a \oplus DMG^{sortie_C}_{out}(l, instr), DMG^{sortie_C}_{in}(l, instr)\}$$

$$avec\, \emptyset \oplus x = \emptyset\, \forall x$$

FIGURE 5.6: Dommage *must* et *pers*

un point p du composant (correspondant par exemple à une instruction de branchement donnée dont on voudra calculer la catégorie) en fonction de l'ABS avant le composant, et des informations de dommage au point p.

Comme pour le cache, une fois que l'on a obtenu les informations de dommage ($DMG^{bb}_{in} \times DMG^{bb}_{out} \times DMG^{bb}_{may} \times ABS^{b}_{may}$) au bloc de base *bb* on peut exprimer la catégorie conditionnelle de l'instruction de branchement *instr* se trouvant à la fin du bloc de base *bb*, sous la forme suivante :

$$CC^{instr} = \lambda ABS^{entrée_C}_{pers} \lambda ABS^{entrée_C}_{must} \lambda ABS^{entrée_C}_{may}.CAT(ABS^{bb}_{may}, ABS^{bb}_{must}, ABS^{bb}_{pers})$$

Les valeurs ABS^{bb}_{may} et ABS^{bb}_{pers} sont calculées par les fonctions $dommagePartiel_{may/pers}$.

Pour la prise en compte de la *persistence* paramétrique, nous enrichissons la fonction *résumé* comme dans le cas du cache : si la catégorie résultante de la fonction résumé est *First Unknown*, alors on réalise un traitement spécifique pour gérer la *persistence* paramétrique (*bb* désigne le bloc de base qui contient *instr*) :

1. L'instruction *instr* est dans (au moins) une boucle du composant, et la boucle la plus externe du composant contenant *instr*, notée L_c, est telle que *instr* n'est pas *persistent* par rapport à L_c La catégorie de *instr* est alors déterminée par ABS^{bb}_{multi}, avec l'aide de la fonction *IsPers()* comme décrit dans la section 5.2.1.2.
2. L'instruction *instr* n'est pas dans une boucle dans le composant, ou bien la boucle la plus externe du composant contenant *instr*, notée L_c, est telle que *instr* est *persistent* par rapport à L_c. L'$ABS^{avant_C}_{multi}$ contient n éléments, correspondant aux n boucles de L_0 à L_{n-1} qui contiennent l'appel au composant. Pour chaque boucle L_i, on a $ABS^{bb}_{multi}[L_i] = dommagePartiel_{pers}(ABS^{entrée_C}_{multi}[L_i], DMG^{bb}_{in}, DMG^{bb}_{out})$. Une fois obtenus les n éléments de $ABS^{bb}_{multi}[L_0]$ à $ABS^{bb}_{multi}[L_{n-1}]$, on calcule la catégorie de *instr*.

5.2.2.3 Composition

La composition de l'analyse partielle de prédiction de branchement se réalise de manière similaire à celle du cache : lorsqu'on analyse le programme principal, chaque appel au composant est représenté par un nœud dont la fonction *Update* est donnée par la fonction de dommage du composant (après avoir réalisé la translation d'adresses liée à l'adresse d'implantation du composant). Ainsi, on peut obtenir les ABS pour l'ensemble du programme principal.

Une fois ceci effectué, on connaît les ABS à chaque entrée de composant, donc la fonction résumé est utilisée (ajustée de manière similaire à la fonction de dommage en raison de l'adresse d'implantation du composant) pour trouver les catégories des instructions de branchement des composants. Pour les instructions de branchement dont les catégories sont *Always D-Predicted* ou *Not Classified*, il est facile de générer les contraintes ILP de la manière décrite dans la section 5.2.1.3.

Toutefois, si une instruction de branchement *instr* est catégorisée *First Unknown*, alors il faut rechercher les autres instructions ayant la même adresse que *instr*, catégorisées *First*

Unknown par rapport à la même boucle. Ces instructions seront forcément elles aussi dans le même composant (mais peut-être dans une instance différente). Donc, on doit rechercher dans tous les appels au composant qui se trouvent dans la boucle associée à la catégorie de *instr*, et identifier toute les instructions ayant la même adresse et catégorie que *instr*. Ceci est possible car la fonction *résumé* indique quelles instructions du composant possèdent la même adresse.

Chapitre 6

Le *Program Structure Tree* et les régions

Les analyses partielles présentées jusqu'ici ne concernaient pas IPET. En fin de compte, nous aboutissons toujours, comme dans le cas d'une analyse monolithique, à un système ILP correspondant à toute la tâche (programme principal et composants), qu'il faut résoudre. Dans ce chapitre, nous proposons une amélioration de la méthode IPET qui permet gagner un temps considérable sur la résolution du système ILP grâce au découpage du CFG en régions indépendantes, pour lesquelles des sous-systèmes ILP pourront être générés et calculés indépendamment. Ces sous-régions sont également parfaites pour représenter les composants.

6.1 Contexte et motivation

La méthode IPET, fréquemment utilisée lors du calcul de WCET, modélise le flot de contrôle du programme et les effets matériels sous forme d'un système ILP. La résolution de ce système ILP est coûteuse en temps. Pour essayer de réduire ce temps, nous nous intéressons dans les deux prochaines sections, à la structure de ces systèmes ILP ainsi qu'à l'évolution du temps de résolution ILP en fonction de divers paramètres du système.

6.1.1 IPET : description des systèmes ILP

Avant de rentrer dans les détails de l'approche présentée dans ce chapitre, nous devons avoir une vision d'ensemble de la structure des systèmes ILP générés avec IPET. Il y a un grand nombre de méthodes utilisées pour gérer divers effets (matériels ou logiciels). Nous étudions ici uniquement l'impact des analyses que nous avons présentées dans ce document (cache d'instructions, limites de boucles, prédiction de branchements) sur le système ILP, mais des études similaires peuvent être faites pour d'autres analyses.

6.1.1.1 Contraintes ILP liées au flot de contrôle

L'analyse du flot de contrôle des tâches, telle que présentée dans [53], représente le programme à analyser par un CFG. Avec IPET, une variable ILP est créée pour chaque bloc de base et pour chaque arc, représentant le nombre de fois que l'on exécute le bloc de base ou que l'on emprunte cet arc. On nomme x_i la variable associée au bloc de base i, et on nomme $e_{i,j}$ la variable associée à l'arc allant du bloc de base i vers le bloc de base j. Pour représenter la structure du CFG dans le système ILP, des contraintes spécifiques (appelées *contraintes structurelles*) sont créées : pour chaque bloc de base, la somme du nombre de passages par les arcs entrants doit être égale à celle du nombre de passages par les arcs sortants, et égale aussi au nombre d'exécutions du bloc de base. Par exemple, pour le bloc de base i, la contrainte serait $\sum_{j \in pred_i} e_{j,i} = x_i = \sum_{j \in succ_i} e_{i,j}$.[1]

Des contraintes supplémentaires sont requises pour représenter les bornes de boucles. Chaque boucle doit posséder une borne *max* ou une borne *total,* ou les deux. S'il existe une borne *max*, on doit créer une contrainte $\sum_{i \in backEdgeSet} e_i \leq max \times \sum_{j \in entryEdgeSet} e_j$ pour limiter le nombre d'itérations de la boucle pour chaque entrée dans celle-ci. S'il existe une borne *total*, on doit créer une contrainte $\sum_{i \in backEdgeSet} e_i \leq total$ pour limiter le nombre total d'itérations de la boucle, ainsi qu'une contrainte de type $\sum_{i \in backEdgeSet} e_i \leq total \times \sum_{j \in entryEdgeSet} e_j$ pour tenir compte du cas où la boucle n'est jamais exécutée et éviter que le corps soit considéré exécuté alors que l'arc entrant n'a pas été traversé.

6.1.1.2 Contraintes ILP liées au cache d'instructions

Considérons la technique d'analyse de cache d'instructions de C. Ferdinand et al., associée à la *persistence* améliorée décrite dans le chapitre 3. Nous rappelons que pour chaque l-bloc, une catégorie est associée : *Always Hit*, *Always Miss*, *Persistent*, ou bien *Not Classified.* La catégorie *Persistent* est paramétrée par une boucle L. Pour chaque l-bloc i, j appartenant au bloc de base i, deux variables $x_{i,j}^{hit}$ et $x_{i,j}^{miss}$ sont créées pour représenter le nombre de *hits* et de *miss* associés à ce l-bloc. Pour chaque l-bloc, le nombre total d'exécutions (*hit* + *miss*) est égal au nombre d'exécutions du bloc de base le contenant, donc on a la contrainte $x_i = x_{i,j}^{hit} + x_{i,j}^{miss}$ pour chaque l-bloc.

La catégorie de chaque l-bloc provoquera la génération d'une contrainte, comme précisé pour les catégories *Always Hit, Always Miss,* et *Not Classified* dans la table 2.2. Pour la catégorie *Persistent(L)*, il y a création d'une contrainte $x_i^{miss} \leq \sum_{i \in pred_L \wedge L \overline{dom}\, i} e_{i,L}$ (où $e_{i,L}$ est la variable associée à un arc d'entrée de la boucle, c'est-à-dire tout arc (i, L) tel que L est la tête de boucle de L et L ne domine pas i)

6.1.1.3 Contraintes ILP liées au pipeline

Dans l'article original de Li. et al. [53] décrivant la méthode IPET, le temps d'exécution de chaque bloc de base est considéré comme une constante, trouvée grâce à une

1. Une version optimisée de cette approche permet de se passer des variables x_i en les remplaçant par la somme des arcs des entrants ou celle des arcs sortants.

simulation ou bien une exécution sur un processeur réel, par exemple. Comme précisé dans la section 2.5.3, J. Engblom et al. ont ensuite proposé dans l'article [26] d'utiliser la simulation pour calculer le temps associé à chaque bloc de base, puis de prendre en compte le gain de temps dû au recouvrement des blocs de base voisins dans le pipeline grâce à la méthode des deltas. Toutefois, il a été montré [61, 90] que cette approche n'est pas totalement sûre (conservatrice) en raison des effets longs. C'est pourquoi nous avons préféré l'approche *exegraph* présentée dans l'article [79], qui permet d'estimer le temps d'exécution d'un bloc de base de manière paramétrique, en fonction du contexte.

Si on utilise *exegraph* uniquement pour calculer le temps d'exécution de chaque bloc de base en faisant l'union de tous les contextes possibles, aucune contrainte supplémentaire n'est générée : le temps d'exécution de chaque bloc de base est simplement modifié pour tenir compte des effets de pipeline, ce qui affecte uniquement la fonction objectif. Toutefois, pour plus de précision il est possible de considérer les prédécesseurs de chaque bloc de base : si un bloc de base possède plusieurs prédécesseurs, son temps d'exécution sera différent selon le prédécesseur (à cause des effets de recouvrement dus au pipeline). Pour prendre ceci en compte, on peut par exemple affecter à chaque bloc de base son temps maximal (calculé en prenant en compte l'union de tous les contextes possibles), et ensuite associer à chaque arc d'entrée un gain de temps lié aux effets de recouvrement.

Par exemple, si un bloc de base i possède deux prédécesseurs, u et v, alors les termes $diff_{u,i} \times e_{u,i}$ et $diff_{v,i} \times e_{v,i}$ seront ajoutés à la fonction objectif. Les variables $diff_{u,i}$ et $diff_{v,i}$ sont égales au gain de temps lorsque le flot de contrôle vient, respectivement, du prédécesseur u, et v. On peut généraliser ceci en s'intéressant non plus uniquement aux prédécesseurs, mais aux séquences de prédécesseurs de longueur arbitraire. Ceci nécessite d'introduire, pour chaque séquence $[i, ..., j]$, une variable $seq_{i..j}$ représentant le nombre d'exécutions de cette séquence (il faudra ensuite contraindre cette variable de manière similaire à ce qui est fait pour l'approche des *delta*s, comme expliqué section 2.5.3). Ensuite pour chaque séquence $[i, ..., j]$ on rajoute un terme $diff_{i..j} \times seq_{i..j}$ pour prendre en compte le gain de temps éventuel lié au contexte favorable. On augmente la complexité du système ILP au fur et à mesure qu'on augmente la taille du préfixe, mais cela permet d'obtenir un WCET plus précis.

6.1.2 Caractéristique du temps de calcul ILP

Les dernières sections ont montrées que les systèmes ILP qu'on doit résoudre lors du calcul de WCET peuvent facilement avoir des tailles importantes, en particulier à cause des effets d'architectures. Au final, on a besoin d'une variable pour chaque bloc de base et pour chaque arc, et de deux contraintes structurelles pour chaque bloc de base. On a besoin d'une à trois contraintes supplémentaires par boucle (en fonction de la présence ou non du *max* et du *total*).

Avec la méthode des graphes d'exécution, selon la longueur du préfixe on aura aucun ajout dans le système ILP (préfixe de taille 0), un ajout de termes dans la fonction objectif uniquement (préfixe de taille 1), ou bien un ajout de termes, variables et contraintes

(préfixe de taille 2 ou plus). Pour gérer le cache, il faut une variable et une contrainte pour chaque l-bloc. Pour une tâche comportant n_N blocs de base, n_L boucles, n_E arcs, et en moyenne n_B l-blocs par bloc de base, sachant qu'en principe on a $n_E > n_N$ et en considérant qu'on n'utilise pas de préfixe pour exegraph, on produira un système ILP avec $(1+n_B)\times n_N+n_E$ variables et $(2+n_B)\times n_N$ contraintes. L'utilisation d'un préfixe exegraph de taille 1 rajoutera n_E variables, et les préfixes supérieurs à 1 généreront un très grand nombre de variables et contraintes supplémentaires. Avec des tâches de plusieurs milliers de blocs de base, ceci fait d'énormes systèmes ILP à résoudre, et les choses sont encore pires si on utilise du déroulage de boucles, car on duplique les contraintes et les variables pour les blocs dans des boucles ($\times 2^n$, n étant le niveau d'imbrication de la boucle).

Étant donné que les systèmes ILP ont un temps de résolution en général non linéaire par rapport au nombre de variables/contraintes, cette résolution est parfois très coûteuse, et constitue souvent l'essentiel du temps passé à calculer le WCET. Ceci est illustré dans la figure 1.4, qui montre des mesures de temps de résolution ILP. Ces mesures ont été faites à l'aide de *lp_solve* [60], sur une centaine de morceaux de codes provenant des benchmarks Malärdalen [62]. Ces mesures nous ont permis de remarquer qu'il serait plus efficace de partitionner le système ILP en plus petit morceaux (de 100 à 300 contraintes, par exemple) car la somme du temps de résolution des sous-systèmes serait certainement plus petite que le temps de résolution du système global. Nous expliquons comment réaliser ceci dans la prochaine section.

6.2 L'approche du PST, et les régions

Comme précisé au début du chapitre, nous allons essayer de découper le CFG de la tâche en plus petit sous-graphes, pour pouvoir générer un sous-système ILP par sous-graphe, résoudre chaque sous-système séparément, et faire la somme des WCET calculés pour avoir le WCET total (d'autres approches pour segmenter le problème de calcul de WCET ont été envisagées dans [91, 29]). La difficulté principale dans cette approche est d'avoir un WCET total égal au WCET qui aurait été trouvé par une analyse classique (notre approche ne doit pas ajouter de pessimisme si possible). L'idée pour accomplir ce but est d'isoler des sous-graphes du CFG tel que leur WCET est une constante pour le système ILP englobant. C'est pourquoi l'isolation de ces sous-graphes dépend de la structure du système ILP, et des contraintes générées pour modéliser le flot de contrôle et les effets d'architecture.

6.2.1 Définition des régions SESE

Pour chaque bloc de base, les contraintes structurelles font correspondre la somme du nombre d'exécutions des arcs entrants et sortants avec le nombre d'exécutions du bloc de base. On peut remarquer que si une région du CFG est connectée au reste du CFG uniquement par un arc d'entrée, et un arc de sortie, les seules contraintes qui relient les variables représentant des blocs de la région avec celles représentant les blocs hors de la

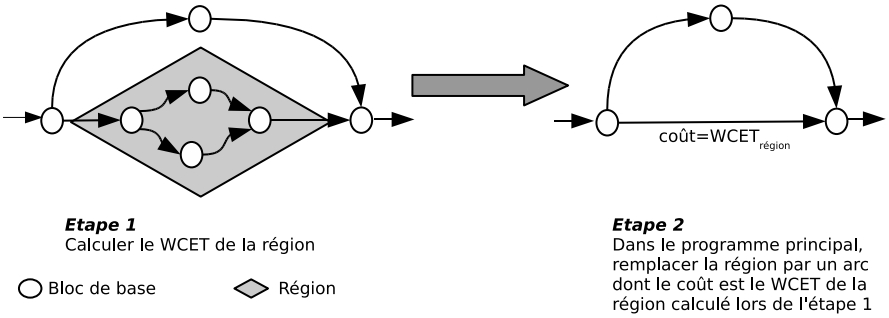


FIGURE 6.1: Utilisation des régions

région sont les contraintes structurelles du bloc de base successeur de l'arc d'entrée de la région (nommé $BB_{entrée}$) et du bloc de base prédécesseur de l'arc de sortie (nommé BB_{sortie}). Ces contraintes sont de la forme $BB_{entree} = e_{arcEntree}$ et $BB_{sortie} = e_{arcSortie}$. De plus, puisque chaque région possède uniquement un arc d'entrée et un arc de sortie, tout le flot de contrôle passant par l'arc d'entrée doit quitter la région par l'arc de sortie, de telle sorte que $x_{bbEntrée} = x_{bbSortie} = e_{arcEntrée} = e_{arcSortie} = x_{région}$, où $x_{région}$ est une variable représentant le nombre d'exécutions de la région.

Si on restreint le système ILP aux contraintes structurelles, on peut en conclure que toute les variables représentant le nombre d'exécutions des blocs de base de la région dépendent, au final, uniquement de x_{region}. En considérant, en plus, les contraintes utilisées pour exprimer le *max* des limites de boucles, cette assertion reste vraie. En effet, ce type de contrainte lie le nombre d'exécutions des arcs retour au nombre d'exécutions des arcs d'entrée, pour chaque boucle. Or, on peut aisément vérifier qu'une boucle dont la tête est dans une région donnée, ne pourra pas avoir un arc d'entrée ou un arc de retour à l'extérieur de cette région. C'est pourquoi nous pouvons calculer localement le WCET de la région (on prends $x_{région} = 1$), pour ce faire il faut construire un système ILP en prenant en compte seulement les contraintes structurelles et les termes de la fonction objectif correspondant aux blocs de base de la région. Lors de l'analyse du programme principal, on peut remplacer la région par un simple arc (dont le coût sera le WCET de la région), enlever du système ILP toute les variables, contraintes et termes de la région, et ajouter le terme $x_{arcRégion} \times WCET_{région}$ à la fonction objectif. Ce procédé est illustré dans la figure 6.1.

Il est important de garder à l'esprit que cette méthode ne conserve le WCET que parce les termes de la fonction objectif qui représentent la contribution au WCET de la région ne dépendent (directement ou indirectement) que de x_{region}. En d'autres termes, cela veut dire que le WCET de la région est toujours le même, quel que soit le contexte d'exécution. Ces régions à un seul arc d'entrée et un seul arc de sortie sont appelées *régions SESE* (Single-Entry Single-Exit) et un algorithme efficace pour leur détection est présenté dans [50].

6.2.2 Prise en compte des divers effets matériels et logiciels avec les régions

Il est intéressant de savoir que si on prend en compte uniquement les contraintes structurelles et les contraintes représentant le *max* des boucles, on peut isoler des régions SESE à WCET indépendant du contexte, car même si la restriction à ce type de contraintes exclut toute gestion d'effets architecturaux (et donc rajoute du pessimisme), on est quand même en mesure de calculer un WCET.

Toutefois, ceci constitue une première étape, et dans les paragraphes suivants nous tenterons d'intégrer la prise en compte des contraintes liées à d'autres effets (boucles avec totaux, cache d'instructions, pipeline, ...)

6.2.2.1 Les limites de boucles

Si une région SESE contient une boucle possédant une limite *total*, alors le WCET de la région n'est plus indépendant du contexte. En effet, le nombre total d'itérations de la boucle est global pour toute l'exécution du programme, dans laquelle pourront avoir lieu plusieurs exécutions de la région. Par exemple, si une boucle dans une région possède un *total* de 15, alors le nombre total d'itérations ne pourra jamais dépasser 15, qu'on exécute la région une seule fois ou bien 10 fois, par conséquent la contribution au WCET de la région n'est pas proportionnelle au nombre de passages par la région.

Une région contenant des boucles de tel type ne peut pas être utilisée afin d'accélérer le calcul du WCET, à moins de modifier la signification du *total* pour qu'il se réfère à un total d'itérations pour chaque entrée dans la région, au lieu d'un total pour tout le programme. Ceci entraîne du pessimisme, mais permet de gagner du temps de calcul.

6.2.2.2 Le pipeline

Nous considérons ici l'analyse de pipeline avec la méthode des graphes d'exécution. Il existe quatre cas différents à prendre en considération : (1) l'utilisation des graphes d'exécution pour calculer le temps du bloc de base en faisant l'union de tous les contextes possibles, (2) le calcul du temps en fonction du prédécesseur, (3) le calcul du temps en fonction d'une séquence de deux prédécesseurs, (4) le calcul du temps en fonction d'une séquence de prédécesseurs de longueur supérieure à deux.

Si on utilise les graphes d'exécution pour calculer le temps de chaque bloc de base en faisant l'union de tous les contextes possibles, le seul impact pour le système ILP est la création de termes $t_i \times x_i$ (où t_i est le temps d'exécution du bloc de base i, calculé comme indiqué précédemment) dans la fonction objectif. Le temps d'exécution de chaque bloc de base est indépendant, et le WCET reste constant pour la région.

Si on calcule le temps de chaque bloc de base en fonction de son prédécesseur, il faudra rajouter dans la fonction objectif les termes $diff_{i,B} \times e_{i,B}$ comme présenté dans la section 6.1.1.3. Le seul bloc de base qui possède un prédécesseur hors de la région est le successeur de l'arc d'entrée de région. Puisque les régions n'ont qu'un seul arc d'entrée, ce

bloc de base n'a qu'un seul prédécesseur, et donc son temps est constant. C'est pourquoi, dans ce cas aussi, le WCET de la région ne dépend pas du contexte.

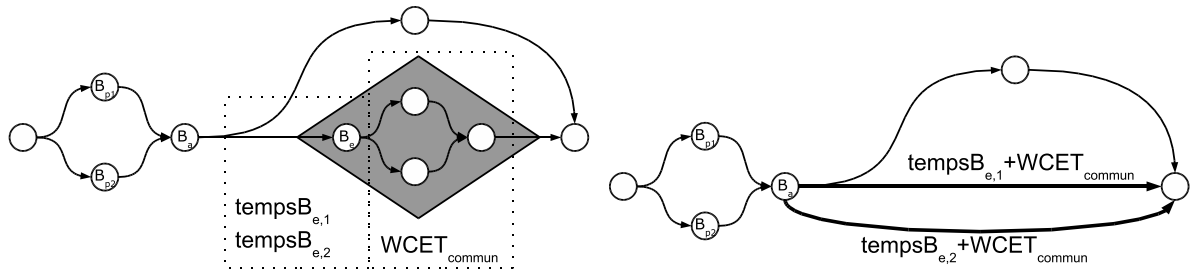
Si on calcule le temps de chaque bloc de base en fonction d'une séquence de prédécesseurs de taille 2, alors le WCET d'une région peut dépendre du contexte. Soit B_a le bloc de base avant l'arc d'entrée de la région, et B_e le bloc de base d'entrée de la région, c'est-à-dire le successeur de l'arc d'entrée de la région. Soient également $B_{p1}, ..., B_{pn}$ les n prédécesseurs du bloc de base B_a. Le bloc B_e possède n possibilités de séquence de prédécesseurs de taille 2, de (B_{p1}, B_a, B_e) à (B_{pn}, B_a, B_e). C'est d'ailleurs le seul bloc qui possède plusieurs possibilités de séquences de prédécesseurs de taille 2 qui ne soient pas contenues dans la région. En effet, pour tout successeur B_s de B_e, la seule séquence de prédécesseurs de taille 2 qui n'est pas contenue dans la région est (B_a, B_e, B_s). Par conséquent il y a n WCETs possibles différents pour la région $(WCET_1, ..., WCET_n)$, selon le prédécesseur parmi $(B_{p1}, ..., B_{pn})$ qui est exécuté. Heureusement il est possible de calculer tous ces WCETs sans devoir faire n calculs de WCET sur la région : en effet, étant donné que le seul bloc de base ayant un temps d'exécution variable est B_e, et qu'on passe par ce bloc exactement une fois par exécution de la région[2], on peut écrire que $WCET_i = WCET_{commun} + temps_{Be,i}$, où $i \in [1..n]$, où $WCET_{commun}$ est le WCET de la région dans lequel le temps d'exécution du bloc B_e est mis à 0, et où $temps_{Be,i}$ est le temps d'exécution de B_e pour la séquence de prédécesseurs commençant par B_{pi}. Comme précisé dans l'exemple de la figure 6.2(a), on peut calculer le terme $WCET_{sansBe}$ lors d'une seule analyse sur la région, puis calculer la valeur des n termes $temps_{Be,i}$. Ensuite, au lieu de remplacer la région par un arc simple comme dans 6.1, on la remplace par n arcs parallèles correspondant aux n $(WCET_1, ..., WCET_n)$, comme montré dans 6.2(b). Les n variables ILP représentant le nombre d'exécutions de ces arcs devront être liées par des contraintes aux nombre d'exécutions des séquences correspondantes, de manière similaire à ce qui a été expliqué dans la section 2.5.3.

Si on utilise une séquence supérieure à deux, alors deux solutions sont possibles pour utiliser quand même les régions. Soit on limite la séquence de prédécesseurs à 2 pour tous les blocs de base de la région qui auraient un morceau de séquence de prédécesseurs à l'extérieur de la région6.2(a), soit on calcule plusieurs fois le WCET de chaque région (un WCET pour chaque séquence de prédécesseurs possibles), tel qu'indiqué dans la figure 6.2(b). Toutefois, les résultats expérimentaux fournis dans [79] montrent que la précision du WCET est bonne en utilisant un seul prédécesseur comme contexte.

6.2.2.3 Le cache d'instructions

Dans la méthode d'analyse de cache avec les catégories, on génère des contraintes différentes pour chaque l-bloc selon que la catégorie est *Always Hit*, *Always Miss*, *Persistent*, ou bien *Not Classified*. Pour savoir si ceci pose un problème par rapport à l'approche des régions, il faut étudier chacun de ces cas.

2. sauf si B_e est une tête de boucle, mais dans ce cas on peut dérouler la première itération de cette boucle et ainsi se ramener au cas faisable



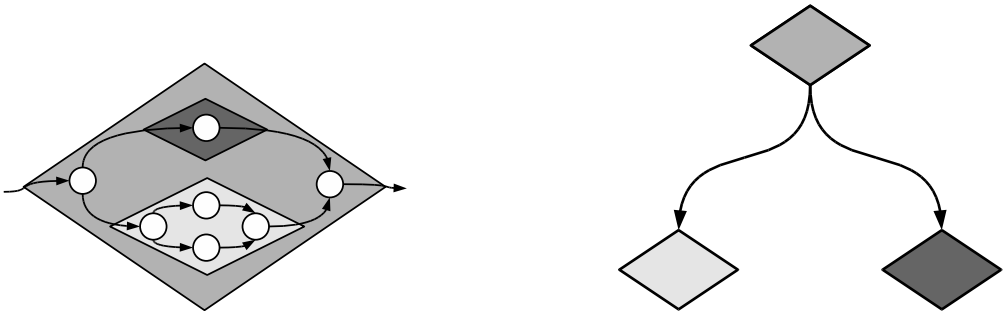
(a) Traitement de la région (b) Utilisation du résultat dans la région parente

FIGURE 6.2: Solutions pour les séquences supérieures à 2

Les catégories *Always Hit* et *Always Miss* signifient que le l-bloc provoquera toujours un hit, ou bien toujours un miss. C'est pourquoi, d'après les contraintes générées, ($x_{i,j}^{miss} = x_i$ et $x_{i,j}^{miss} = 0$), la contribution à la fonction objectif ($x_{i,j}^{miss} \times t_{i,j}^{miss}$) sera proportionnelle à la valeur de x_i (soit égal à x_i, soit égal à 0, ce qui est proportionnel dans les deux cas), qui elle-même est proportionnelle à x_{region}. Donc, la propriété d'indépendance du WCET de la région vis-à-vis du contexte est conservée. Il est à noter que le procédé de déroulage de boucles n'interfère pas, puisqu'il ne fait que dupliquer des parties de code dans la région.

La catégorie *Persistent* peut engendrer des contraintes plus complexes, reliant des variables associées à des blocs de base provenant de plusieurs régions différentes du CFG. La catégorie *Persistent* signifie que la première exécution du l-bloc pourra causer un *miss,* mais que les exécutions suivantes provoqueront obligatoirement un *hit*. La contrainte correspondante est de la forme $x_{i,j}^{miss} \times \sum e_{*,H_l}$ (H_l étant la tête de boucle de L, boucle associée au l-bloc *Persistent*). Si la tête de boucle L est en dehors de la région, alors $x_{i,j}^{miss}$ dépend de quelque chose qui ne peut pas être dérivé du nombre d'exécutions de la région, $x_{région}$. En d'autres termes, et d'un point de vue plus concret, pour chaque exécution de la région, le statut (*hit* ou *miss*) du l-bloc (i, j) dont la catégorie est *Persistent(L)* dépend de la présence ou non du l-bloc dans le cache lors de l'entrée dans la région. Par conséquent, le WCET de la région dépend du contexte.

La catégorie *Persistent* nous montre, encore une fois (après l'exemple d'*exegraph* avec des préfixes de taille supérieure à 2 et des totaux de boucles) que certaines contraintes ILP engendrent des limitations qui empêchent de calculer localement le WCET d'une région. Une région peut être calculée indépendamment si, pour tous les l-blocs *Persistent* de la région, la tête de boucle associée est aussi incluse dans la région. Dans le cas contraire, si on veut à tout prix utiliser ce type de région pour accélérer l'analyse, on peut (1) considérer que les l-blocs de catégorie *Persistent* par rapport à une boucle dont la tête est à l'extérieur de la région seront qualifiés *Not Classified* (ajout de pessimisme), ou bien (2) calculer plusieurs WCET pour la région, un par contexte possible. Cette dernière solution n'est pas vraiment pratique, car elle implique qu'il faut calculer 2^N WCETs différents si la région comporte N l-blocs de catégorie *Persistent* par rapport à une boucle dont la tête est à l'extérieur de la région.



(a) Vue CFG (b) Vue PST

FIGURE 6.3: Du CFG au PST

6.2.2.4 Généralisation

Nous avons vu quelques limitations concernant l'utilisation des régions. D'un point de vue général, on peut dire que le choix des régions est dicté par la nature des contraintes et des termes de la fonction objectif utilisés pour modéliser le contenu de la région. On dit qu'une région est *faisable* (c'est-à-dire calculable de manière indépendante) si (1) les termes de la fonction objectif utilisés pour représenter les blocs et l-blocs de la région sont proportionnels à $x_{région}$, et si (2) toute contrainte contenant des variables associées aux blocs ou l-blocs de la région, contient *uniquement* des variables d'éléments associés à la région (à l'exclusion de toute variable associée à un bloc extérieur, ou toute constante différente de 0).

6.2.3 Algorithme de calcul de WCET avec régions

Nous avons vu que les diverses analyses liées au WCET (en particulier l'analyse du cache d'instructions) pouvaient rendre certaines régions infaisables, et donc limiter notre approche de calcul de WCET par partitionnement. C'est pourquoi, pour augmenter l'efficacité de notre méthode, nous proposons de compenser l'apparition de régions infaisables en organisant les régions sous forme d'arbre afin de hiérarchiser notre approche de calcul de WCET. Cette forme d'arbre est appelée PST (*Program Structure Tree*).

Un PST est un arbre dont les nœuds sont des régions SESE canoniques. Une région SESE canonique est une région SESE qui ne contient pas d'autre régions SESE partageant ses arcs d'entrée ou de sortie. Cette propriété est importante car cela empêche les régions SESE canoniques de se chevaucher, ce qui permet leur organisation en arbre. Dans le PST, le nœud racine correspond à la région SESE représentant tout le programme. Les fils d'un nœud sont les régions incluses. Les feuilles représentent des régions minimales. L'article [50] présente un algorithme efficace pour calculer le PST d'un programme en temps linéaire. A titre d'exemple, la figure 6.3(a) montre un CFG, et la figure 6.3(b) montre le PST correspondant.

Pour calculer le WCET sur un programme dont le PST a été construit, on fait un

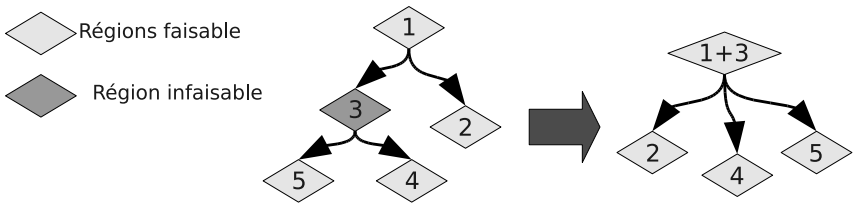


FIGURE 6.4: PST faisable

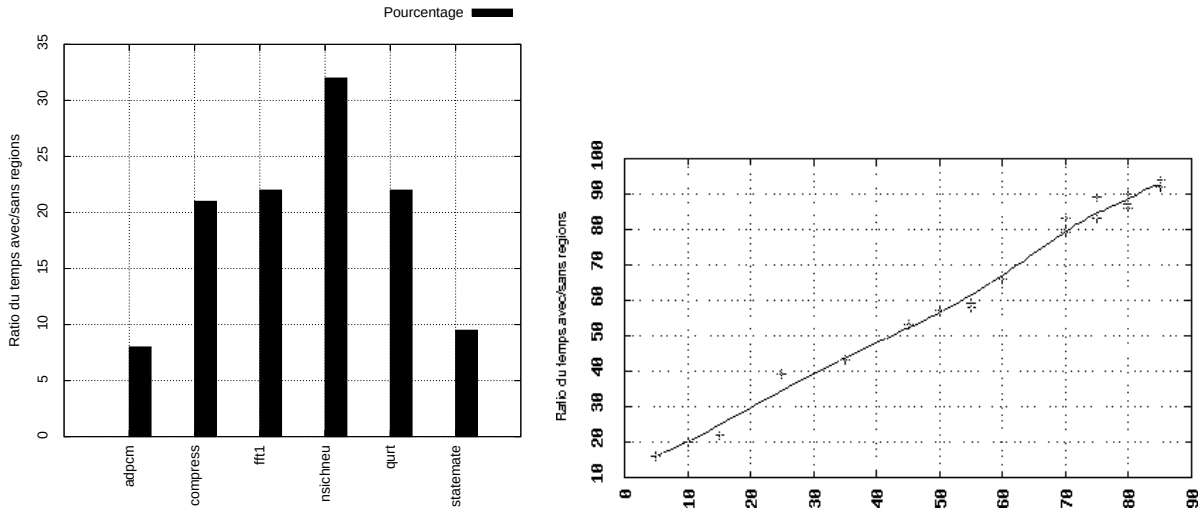
parcours en profondeur d'abord du PST, en partant des feuilles, et en allant jusqu'à la racine. Pour chaque région traitée, on calcule son WCET, puis on la remplace par un arc dont le coût est le WCET que l'on vient de calculer, ce qui élimine la région de l'arbre et permettra de calculer plus facilement le WCET de la région parente. On procède ainsi de suite jusqu'à aboutir au WCET de la région racine, c'est-à-dire le WCET du programme complet.

Bien sûr, le PST n'est pas utilisable tel quel, à cause des régions infaisables. Le processus de construction du PST doit être modifié pour éliminer les régions infaisables, comme décrit dans la figure 6.4. Pour ce faire, le PST est parcouru des feuilles vers la racine. Lorsqu'une région infaisable est trouvée, elle est enlevée, et ses blocs et régions filles sont affectés à la région parente. Le résultat est un arbre dont les nœuds sont les régions faisables, et qui respecte la structure du PST original. Pour calculer le WCET, on procède comme décrit dans le paragraphe précédent, à ceci près qu'on utilise le PST modifié.

Certains logiciels de résolution ILP sont moins efficaces sur de très petits systèmes. En d'autre mots, pour un système ILP correspondant à une « grosse » région de taille M, il est plus rapide de résoudre séparément deux régions de taille $\frac{M}{2}$, toutefois, pour un système correspondant à une « petite » région de taille N, il sera plus rapide de résoudre le système ILP dans son ensemble. Ceci est dû au fait que certains systèmes ILP ont un temps de résolution de la forme $temps = K + f(taille)$, où $f(taille)$ est un temps fonction de la taille du système ILP (la plupart du temps cette fonction croît de manière non linéaire), et où K est une constante, un temps inévitable qui ne dépend pas de la taille du système. Selon le logiciel de résolution ILP utilisé, il existe donc une taille limite en dessous de laquelle il n'est pas intéressant d'avoir des régions. Dans notre approche de construction du PST faisable, nous pouvons donc considérer comme infaisable les régions qui contiennent moins de blocs de base qu'un certain seuil (dépendant du logiciel de résolution ILP), ces régions seront alors fusionnées avec leur région parente.

6.3 Expérimentation

Nous avons testé notre approche sur OTAWA, le solveur de système ILP lp_solve, avec une architecture cible possédant un processeur à pipeline simple, et un cache d'instructions. Les effets de pipeline ont été pris en compte avec la méthode des graphes d'exécution. Les mesures ont été faites sur un sous-ensemble des benchmarks Malärdalen.



(a) Gain de temps grâce aux régions (b) Impact des dépendances

FIGURE 6.5: Régions et gain de temps

Nous avons choisi d'utiliser uniquement les tests assez gros, c'est-à-dire suffisamment gros pour pouvoir extraire des régions (des tests générant des systèmes ILP de plus de 300 contraintes).

6.3.1 Comparaison du temps d'analyse

Pour chaque test sélectionné parmi les benchmarks Malärdalen, le WCET a été calculé d'abord avec l'approche traditionnelle, puis ensuite avec l'approche de partitionnement par régions. Le temps d'analyse de flot et d'effets architecturaux a été retranché du temps total, car il est le même dans les deux approches. Dans le cas de l'analyse avec partitionnement, nous tenons compte du temps passé à réaliser le partitionnement (construction du PST, identification des régions infaisables, etc ...)

En moyenne, notre approche de partitionnement permet de faire l'analyse 6.5 fois plus vite. Des détails sont fournis dans la figure 6.5(a), qui donne, pour chaque test, le ratio $\frac{temps_{notreApproche}}{temps_{approcheTraditionnelle}}$ (sous forme d'un pourcentage, par exemple 33% signifie que l'approche avec régions prends un tiers du temps pris par l'analyse traditionnelle), dans le cas du déroulage de boucle, et sans déroulage de boucles.

6.3.2 Ajout de dépendances aléatoires

Dans les sections précédentes, nous avons déterminé que la prise en compte de certaines analyses impliquées dans le WCET peut créer des dépendances entre des blocs qui appartiennent à des régions différentes, ce qui peut rendre certaines régions infaisables. Comme nous ne pouvons pas modéliser les dépendances liées à toutes les analyses de WCET existantes (sans parler de celles qui n'existent pas encore), nous avons décidé de

tester la robustesse de notre approche en introduisant une certaine quantité de dépendances aléatoires supplémentaires.

Ces dépendances généralisent le concept de dépendance inter-régions telle que la *persistence* dans l'analyse de cache d'instructions : un bloc *persistent* est lié à sa tête de boucle par une contrainte. En d'autre termes, une dépendance allant d'un bloc de base source BB_s à un bloc de base destination BB_d représente le nécessité d'avoir des informations sur le nombre d'exécutions de BB_s pour connaître le temps d'exécution de BB_d. Cette propriété peut être appliquée à de nombreux types d'analyses, en dehors de la catégorisation des l-blocs lors de l'analyse du cache. Nous avons utilisé un générateur aléatoire de dépendances pour simuler des analyses supplémentaires pouvant perturber notre algorithme par la création de régions infaisables.

La figure 6.5(b) montre l'impact de ces dépendances supplémentaires sur le temps d'analyse. L'axe des abscisses représente le pourcentage de blocs de base affectés par (au moins) une dépendance supplémentaire (c'est-à-dire que ce pourcentage ne tient pas compte des dépendances réelles dues par exemple à la *persistence*). L'axe des ordonnées représente le temps moyen d'analyse (pour les même tests que dans la section précédente). Les résultats montrent que même si un bloc de base sur deux est affecté par une dépendance (ce qui constitue un nombre important de dépendances), notre approche permet d'obtenir quand même un gain de temps significatif, en moyenne 1.7 fois plus rapide sur le sous-ensemble des benchmarks Malärdalen utilisés dans la figure 6.5.

6.4 Perspectives concernant les régions

Ce chapitre a présenté une méthode de subdivision du CFG en régions indépendantes pour obtenir des systèmes ILP plus petits, et donc plus faciles à résoudre, dans le but d'augmenter la vitesse de la méthode IPET. Nous avons exploré l'impact de diverses analyses sur l'inter-dépendances des régions, et les expérimentations que nous avons réalisées avec OTAWA montrent que notre méthode est efficace, et ce même en présence d'un nombre important de dépendances.

L'approche des régions étudiée dans ce chapitre marque un certain changement par rapport aux méthodes présentées dans les chapitres précédents (analyse partielle du cache et des limites de boucles, avec fonctions *transfert*, *résumé*, etc...), en particulier parce qu'elle ne sert, pour le moment, qu'à faire gagner du temps à la méthode IPET et non à traiter le calcul de WCET sur des composants. Toutefois, introduire ces concepts de PST et de régions est utile, car ils seront à nouveau utilisés lorsque nous généraliserons la méthode d'analyse partielle des composants, dans le chapitre suivant.

Chapitre 7

Généralisation de l'analyse partielle

Jusqu'ici, nous avons vu plusieurs méthodes d'analyse partielles affectant diverses phases du calcul du WCET (analyse partielle du cache et des limites de boucles, approche des régions, etc...). Nous n'avons pas pour le moment de technique d'analyse partielle de composant fonctionnant de bout en bout : l'analyse partielle de cache sait produire son résultat partiel, l'analyse partielle des limites de boucles aussi, mais il n'y a pas d'intégration. Dans ce chapitre, nous allons proposer une approche générale, en nous basant sur les éléments déjà présentés dans les chapitres précédents.

7.1 Notre approche générique d'analyse partielle

Cette partie décrit notre approche d'analyse partielle de WCET. Dans un premier temps, nous définissons le contexte d'utilisation de notre méthode, puis le contenu de nos résultats partiels. Nous expliquons ensuite comment ces résultats partiels sont produits et utilisés, puis comment cette méthode peut être optimisée par l'utilisation de régions. Nous illustrons ceci grâce à un exemple ainsi que des résultats expérimentaux.

7.1.1 Définition du problème, contexte d'utilisation

Considérons un exemple : un composant est développé par une entité (appelée, pour le besoin de l'exemple *Producteur*). Ce composant est un composant "boîte noire", et il est délivré à une entité (appelée, pour les besoins de l'exemple *Assembleur*) dont le rôle est de développer le programme principal, et de le lier au composant (entre autres) pour créer un exécutable complet. Le programme principal possède des tâches temps réel, et donc leur WCET doit être connu. Même si l'Assembleur n'a aucun accès au code du composant, ce composant doit être analysé : le Producteur doit calculer le résultat partiel du composant et le livrer avec le code binaire de celui-ci. Avec ces informations, l'Assembleur pourra calculer le WCET de son application (figure 7.1).

Un composant peut être défini comme un bloc de code avec un ou plusieurs points d'entrées. Quand une analyse est réalisée, il y a une analyse partielle pour chaque point d'entrée de composant, chacune donnant un résultat partiel. L'agrégation de ces résultats

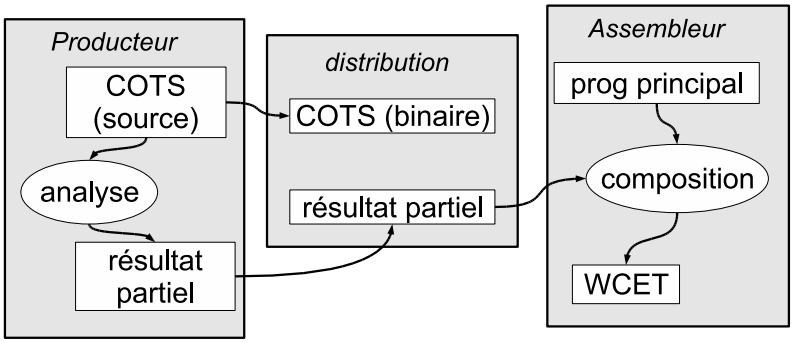


FIGURE 7.1: Modèle de distribution

partiels est le résultat partiel du composant.

7.1.2 Résultat partiel

Le résultat partiel d'un point d'entrée de composant doit contenir toute les informations nécessaires pour permettre d'analyser le programme principal ; en particulier il doit contenir les résultats partiels de toutes les analyses liées au WCET (cache, prédiction de branchements, limites de boucles, etc...).

Dans les chapitres 4 et 5, nous avons pu remarquer que l'analyse partielle pouvait être décomposée en trois sous-problèmes : (1) le fait que l'appel au composant a un impact sur le temps d'exécution des blocs de base du programme principal, (2) le fait que le temps d'exécution de l'appel au COTS dépends du contexte d'appel, et (3) la variabilité de l'adresse d'implantation du composant lors de l'édition de liens.

Un bon nombre d'analyses liées au calcul du WCET (notamment celles utilisant l'interprétation abstraite) manipulent et produisent des états abstraits en chaque point du programme. Ces états abstraits sont ensuite utilisés pour produire un résultat ou une propriété utile pour le calcul du WCET. Par exemple, les techniques d'analyse de cache d'instructions par catégorisation (voir les articles [34, 32] et [67, 66] pour deux méthodes différentes), d'analyse de limites de boucles [24], ou d'analyse de prédicteur de branchements [15] rentrent dans cette catégorie.

Pour ce type d'analyses, le sous-problème (1) est pris en compte par une fonction *transfert,* et le sous-problème (2) est géré par une fonction *résumé.* Ces fonctions seront en fait des généralisations des fonctions *transfert* et *résumé* que nous avons présentées pour les analyses des sections 4 et 5, notamment avec quelques extensions concernant la contribution du composant au système ILP).

Pour traiter le troisième sous-problème, il faudra tenir compte des informations de position lors de l'analyse partielle et de la composition.

7.1.2.1 Fonctions transfert

La généralisation de la fonction *transfert* est relativement simple : cette fonction est de la forme $etat^{après} = transfert(etat^{avant})$, et donne l'état après l'appel au composant, en fonction de l'état avant. Les termes $etat^{après}$ et $etat^{avant}$ sont du même type, c'est-à-dire

le type de l'état abstrait, spécifique au domaine de l'analyse. Si le domaine de l'analyse n'est pas le même pour le composant et pour le programme principal (par exemple pour le cache, la plage des valeurs autorisées dans les ACS ne sera pas la même car les blocs de cache du composant ne correspondent pas tous à des blocs de cache du programme principal), alors la fonction *transfert* doit être extensible (adaptable) de telle sorte que, lorsqu'on tente de l'utiliser lors de la composition, sa signature correspond au domaine utilisé pour le programme principal. Cette dernière constatation est aussi vraie pour la fonction *résumé.*

7.1.2.2 Fonctions résumé

Pour la généralisation de la fonction *résumé*, il faut tenir compte des deux étapes que la plupart des analyses statiques liées au WCET emploient : (1) le calcul d'états abstraits en tout point du programme, et (2) l'utilisation de ces états abstraits pour produire le résultat final de l'analyse, tel que les catégories. Le type de l'état abstrait, ainsi que la méthode pour en déduire le résultat final sont spécifiques à l'analyse. Toutefois, toutes les analyses contribuent au système ILP d'une façon ou d'une autre : il doit être possible de généraliser ceci.

Pour ce faire, nous modélisons la contribution du composant au WCET par un système ILP paramétrique. Le résultat de la fonction *résumé* est l'instanciation de ce système paramétrique, à partir des valeurs des paramètres. Les paramètres sont des valeurs entières (pouvant avoir des noms symboliques si on veut avoir un résultat partiel lisible). Dans le système ILP paramétrique, chaque terme appartenant à une contrainte peut optionnellement comporter un facteur paramètre (exemple : $2x_1.par_1 + 3.x_2 = 5.par_2$). On peut aussi donner la possibilité d'ajouter des contraintes qui ne seront activées que sous certaines conditions. Par exemple, si la catégorie du l-bloc 1 est donnée par le paramètre par_1, alors on peut rajouter une contrainte conditionnelle $if(par_1 = AH)\ \{x_1^{miss} = 0\}$.

Pour chaque point d'entrée de composant et pour chaque analyse, le résultat partiel comporte un ensemble de paramètres représentant des propriétés utiles pour calculer le WCET et dépendantes du contexte d'appel. Chaque contexte d'appel possible peut être représenté par un jeu de valeurs de ces paramètres, ces valeurs permettent d'instancier le système paramétrique représentant la contribution du composant au WCET. La fonction *résumé* prend $etat^{avant}$ en paramètre, et retourne une liste de paires (nom, $valeur$), une paire par paramètre : $< (par_1, val_1, ..., par_n, val_n) >= résumé(état^{avant})$, où par_i sont les noms des paramètres et val_i leurs valeurs. Le système ILP paramétré contient des parties communes (comme les contraintes structurelles), et des parties spécifiques à des analyses (en d'autres mots, les contributions au système de chaque analyse, comme les contraintes de modélisation du cache).

7.1.2.3 Informations de position et relocation

Comme le code du composant est relogeable, l'adresse d'implantation du composant peut varier, on peut admettre qu'elle est connue uniquement au moment de l'édition des

liens. Pour certaines analyses, l'adresse d'implantation a un effet sur le WCET. On peut citer comme exemple l'analyse partielle du cache d'instructions, comme expliqué dans la section 4.2.2 l'adresse d'implantation a un effet sur le découpage des l-blocs.

Pour résoudre ce problème, deux mécanismes peuvent être utilisés (il est aussi possible de les combiner). Premièrement, le résultat partiel peut contenir des informations qui vont restreindre l'ensemble des adresses d'implantations acceptables du composant. Ces restrictions ont pour but de faire en sorte que l'adresse d'implantation soit choisie de manière à ce que le résultat partiel reste utile pour décrire la contribution du composant au WCET lors de la composition. Cette contrainte est illustrée dans l'analyse du cache d'instructions, lorsqu'on impose que l'adresse d'implantation soit un multiple de la taille du bloc de cache.

Deuxièmement, si une analyse est affectée par l'adresse d'implantation du composant, les fonctions *transfert* et *résumé* doivent prendre ces adresses en paramètre. C'est-à-dire que la fonction *transfert* est définie par $état^{après} = transfert(adresse, etat^{avant})$ (et la fonction *résumé* est étendue de manière similaire). Là aussi, ceci est utile pour l'analyse de cache : pour pouvoir appliquer la fonction *transfert* sur l'ACS d'entrée, il faut d'abord pratiquer un décalage sur le dommage de cache du composant, ce décalage dépendant de l'adresse d'implantation.

7.1.2.4 Représentation XML du résultat partiel

Dans cette section, nous avons défini ce que devait contenir le résultat partiel. Toutefois, pour que les analyseurs soient capables de communiquer entre eux, il est nécessaire de définir un format de stockage des résultats partiels associés aux composants, prenant en compte la fonction *transfert*, la fonction *résumé*, le système ILP paramétrique associé au composant, et les informations de position et de relocation. Ce format doit être assez extensible (pour des analyses futures), tout en restant assez spécifique pour être utile. Nous avons réalisé la conception d'un tel format, nous avons choisi de l'exprimer en XML pour des raisons d'extensibilité et de flexibilité.

Les divers éléments du résultat partiel présentés dans les sous-sections précédentes sont intégrés sous forme d'un arbre d'éléments XML et d'attributs. Certaines informations du résultat partiel (comme le système ILP paramétré) sont suffisamment génériques pour pouvoir être spécifiées précisément dans le format, tan disque d'autres sont spécifiques à une certaine analyse (comme le vieillissement des blocs de cache), ces dernières devront être représentées par un élément XML opaque, décodable uniquement par l'analyseur qui nécessite cette information.

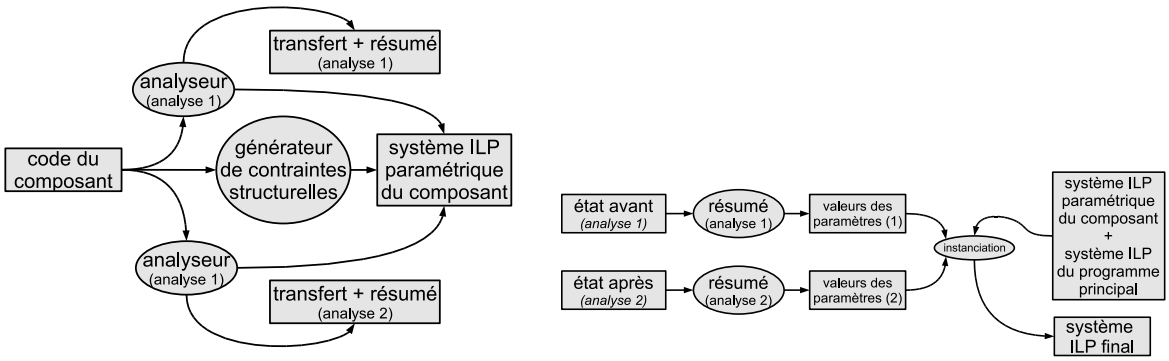
Un composant peut contenir plusieurs points d'entrée (c'est-à-dire plusieurs fonctions appelables depuis l'extérieur), chaque point d'entrée doit avoir son résultat partiel indépendant. Pour chaque point d'entrée de composant, il y a deux parties : (1) pour chaque analyse, les informations opaques représentant la fonction *transfert* et la fonction *résumé*, et (2) au niveau global, le système ILP paramétré représentant la contribution du composant au WCET (ce système contient des contributions de chaque analyse, mais le résultat

est fusionné).

L'élément racine *component* contient une collection de points d'entrée, chacun représenté par un élément *function*. Chaque élément *function* correspond à une fonction spécifique dans le composant, et contient le résultat partiel associé. Les fonctions qui ne sont pas des points d'entrée (c'est-à-dire qui sont appelables uniquement par d'autres fonctions du composant) ne doivent pas apparaître dans la liste, elles seront prises en compte dans les résultats partiels des points d'entrée depuis lesquelles elles seront appelées. A l'intérieur de l'élément *function*, on a un élément *analysis* pour chaque analyse liée au WCET prise en compte dans ce résultat partiel (par exemple l'analyse du cache d'instructions ou l'estimation des bornes de boucles). Pour chaque élément *analysis*, les fonctions *transfert* et *résumé* correspondantes sont représentées respectivement par des éléments XML *transfer* et *summary*. Bien que le contenu de ces éléments soient dépendant de l'analyse, XML nous permet de les traiter facilement en tant que données opaques. Le système ILP est stocké dans chaque élément *function* (et non pas dans *analysis*, car le système est global à toutes les analyses) sous forme d'un élément XML fils *system*. L'attribut *entry* désigne la variable ILP qui représente le nombre d'exécutions du bloc de base d'entrée du composant. Cette information est nécessaire pour fusionner le système ILP du composant avec le système ILP principal. En effet il est nécessaire de relier le nombre d'exécutions du composant au nombre d'appels de celui-ci, pour ce faire il faudra ajouter une contrainte $x_{entréeComp} = \sum e_{appelComp}$ (où $x_{entréeComp}$ est la variable ILP désignée par l'attribut *entry*, et où $e_{appelComp}$ est la variable correspondant à l'arc d'appel du composant dans le système ILP principal). Dans le système, la fonction objectif est représentée par un élément *objective*, comportant un attribut *const* représentant la partie constante. Les contraintes sont représentées par des éléments *constraint,* qui comportent des attributs *op* et *const* pour représenter respectivement le type de comparateur (égalité, plus-grand-que, etc.) et la constante. La fonction objectif et les contraintes contiennent des termes représentés par des éléments *term*. Ces termes comportent des attributs *coef*, *var*, et *param*. Ce dernier attribut est utilisé en cas de dépendance à un paramètre. L'attribut *coef* est obligatoire, mais il est possible d'omettre *var* ou *param* (mais pas les deux, car cela voudrait dire que le terme est constant, et donc il devrait être ajouté à la valeur de l'attribut *const*).

Si l'attribut *param* est présent, alors ce terme est paramétré, son instanciation est traitée comme suit : une fois la valeur du paramètre connue, le coefficient final du terme est obtenu en multipliant l'attribut *coef* et la valeur du paramètre. Si *var* n'est pas présent, la prise en compte du paramètre rend le terme constant, celui ci est donc supprimé et sa valeur est ajoutée à l'attribut *const*.

Les paramètres présents dans les termes permettent de modifier des coefficients ou des constantes, toutefois ces paramètres peuvent être aussi utilisés pour définir des contraintes conditionnelles. Les éléments XML *if* et *switch* permettent de réaliser ceci : un élément *if* contient une condition qui est exprimée grâce aux attributs *param*, *const*, et *op,* ainsi que des élément XML fils *then* et *else* (facultatif). L'élément *switch* contient un attribut *param* spécifiant un paramètre à tester, tandis que chaque contrainte conditionnelle est



(a) Création (b) Utilisation

FIGURE 7.2: Création et utilisation du résultat partiel

stockée dans des éléments XML fils appelés *case*. Chaque *case* correspond à une valeur de paramètre (spécifiée par l'attribut *value*). Ceci est utile par exemple pour le cache, afin de générer conditionnellement des contraintes ILP concernant un l-bloc en fonction de sa catégorie.

Un exemple d'utilisation de ce format XML sera fourni dans la section 7.1.5.

7.1.3 Création et utilisation du résultat partiel

Pour résumer le tout, le résultat partiel du composant contient, pour chaque point d'entrée de composant :

1. pour chaque analyse, une fonction *transfert* prenant en entrée l'état de l'analyse avant l'appel (et potentiellement l'adresse d'implantation) et retournant l'état après
2. pour chaque analyse, une fonction *résumé,* prenant en entrée l'état de l'analyse avant l'appel (et potentiellement l'adresse d'implantation) et retournant la valeur des paramètres du système ILP
3. globalement, le système ILP paramétrique décrivant l'effet de ce point d'entrée de composant sur le WCET.

Comme indiqué dans la figure 7.2(a), chaque analyseur produit ses propre fonctions *transfert* et *résumé* ainsi que sa contribution au système ILP paramétrique, qui contient aussi une partie commune contenant principalement les contraintes structurelles. Pour instancier ce résultat partiel et trouver la contribution au WCET du composant, nous devons :

1. analyser le programme principal en modélisant les appels au composant par la fonction *transfert* adéquate
2. appliquer la fonction *résumé* pour chaque analyse, afin d'obtenir les valeurs des paramètres
3. les paramètres nous permettent d'instancier le sous-système ILP du composant, qui est ensuite fusionné avec le système ILP du programme principal, ce qui nous permet d'obtenir le WCET.

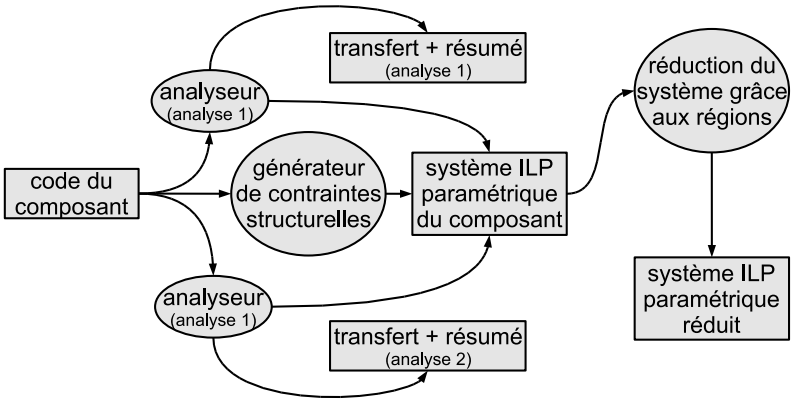


FIGURE 7.3: Calcul du résultat partiel avec régions

L'utilisation du résultat partiel est illustré dans la figure 7.2(b).

7.1.4 Utilisation des régions et minimisation du système

Dans les paragraphes précédents, nous avons vu qu'un système ILP paramétré devait être construit et retourné dans le résultat partiel du composant. Naturellement, ce système peut être calculé comme si on traitait un programme standard : construire les contraintes structurelles, et ensuite rajouter les contraintes liées aux diverses analyses impliquées dans le WCET. La seule différence (du point de vue de la génération de contrainte) d'une telle situation avec le cas d'un programme standard est que l'on doit faire attention aux parties qui dépendent du contexte, et donc introduire des paramètres pour gérer ces dépendances. Par exemple, les contraintes de cache pourraient être construites tout à fait traditionnellement, tout en laissant des contraintes conditionnelles pour les blocs de base qui ont des catégories dépendantes du contexte.

Toutefois, cette approche n'est pas très efficace : le système paramétrique ainsi construit, une fois instancié et fusionné avec celui du programme principal, donnera un système ILP aussi volumineux que celui qui aurait été construit lors d'une analyse monolithique. Donc cette approche a le mérite de rendre le traitement des composants possible, et de gagner du temps sur les diverses analyses impliquées dans le WCET, mais la résolution ILP, qui est une tâche très coûteuse du calcul, prendra toujours autant de temps.

Nous pouvons constater que pour un point d'entrée de composant donné, certaines parties du système ILP ne dépendent pas du contexte d'appel. Pré-calculer ces parties constantes permettrait de minimiser la taille du système ILP paramétré associé au point d'entrée du composant, et d'accélérer la résolution du système lors de la composition. C'est ici que nous pouvons réintroduire les concepts de régions SESE et de PST qui ont été présentés dans le chapitre 6. Nous avions déterminé dans le précédent chapitre que si une région SESE est indépendante du contexte (région faisable), alors son WCET peut être calculé séparément, puis la région remplacée par un arc dont le coût est le WCET de la région. En pré-calculant de la sorte certaines régions du composant, on pourra réduire la taille du système ILP qui devra être incorporé dans le résultat partiel.

```
extern int x;
int fonction(int y, int *tab) {
  int r = 0;
  if (y < 0) {
    r = -1;
  } else {
    do {
      r = r + tab[i];
      i += 3;
    } while (i < y)
  }
  x = x + 10;
  return r;
}
```

FIGURE 7.4: Code source

Pour pré-calculer une région SESE dans ce but, deux conditions doivent être vérifiées :

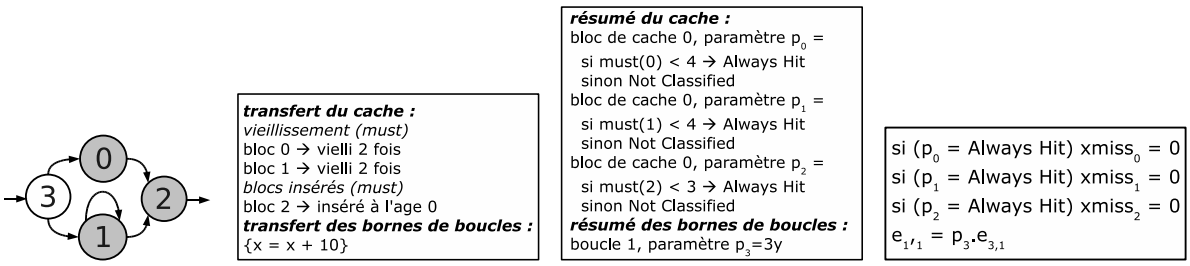
1. la région doit être une région faisable (comme défini dans le chapitre 6)
2. le WCET de la région ne doit pas dépendre du contexte d'appel du point d'entrée du composant, et ce pour toute analyse partielle réalisée sur le composant. Pour l'analyse partielle de cache ou de prédiction de branchement, cela signifie que la région ne doit pas comporter de l-bloc ou d'instruction à catégorie conditionnelle dépendante du contexte. Pour l'analyse des bornes de boucles, cela signifie que la région ne doit pas contenir de boucles dont la limite est paramétrée. D'autres restrictions pourront être apportées par de futures méthodes d'analyses partielles, prenant en compte des effets non traités aujourd'hui.

Grâce à cette méthode (le processus amélioré de création du résultat partiel est montré dans la figure 7.3), nous pouvons calculer tout ce qui peut l'être au moment de l'analyse partielle, et laisser dans le résultat partiel uniquement ce qui dépend du contexte.

7.1.5 Exemple

Soit le code source de la figure 7.4, la figure 7.5(a) montre un exemple simple de CFG correspondant (pour les besoins de l'exemple, les blocs de base de ce CFG sont des l-blocs), où les l-blocs gris appartiennent à la ligne courante de l'analyse (ligne l_0), alors que les l-blocs blancs sont dans une ligne différente. De plus, la borne de la boucle 1 dépend d'un argument d'entrée appelé y, et il y a une variable globale x qui est incrémentée de 10 par l'appel au composant. L'analyse partielle de ce composant produit les fonctions *transfert* et *résumé*, ainsi que le système ILP paramétré, respectivement montré dans les figures 7.5(b), 7.5(c), et 7.5(d).

Les effets du composant sur le cache pour l'ACS *must* est détaillée (avec les informations de vieillissement et la liste des blocs insérés, ainsi que leurs âges) dans la fonction



(a) CFG (b) Transfert (c) Résumé (d) Système ILP

FIGURE 7.5: Exemple pour l'analyse partielle

transfert, sur la figure 7.5(b). L'effet du composant sur la variable globale x est aussi exprimé. La fonction *résumé* dans la figure 7.5(c) définit quatre paramètres, numérotés de p_0 à p_3. Les paramètres p_0, p_1 et p_2 représentent les diverses catégorisations possibles pour les l-blocs 0, 1, et 2. Le paramètre p_3 représente la limite de la boucle, en fonction de l'argument y. La fonction *résumé* donne la valeur des paramètres en fonction des ACS d'entrée (cache) et de y (limites de boucles). La figure 7.5(d) montre la partie paramétrique du système ILP du composant (les parties constantes ne sont pas montrées car cela présenterait peu d'intérêt ...). Il y a une contrainte conditionnelle pour chaque paramètre : pour chaque l-bloc Always-Hit on génère une contrainte qui fixe x_{miss} à 0, et enfin la dernière contrainte modélise la limite de boucle de manière paramétrique, dépendante de p_3.

Pour réaliser la composition, il faut d'abord exécuter les analyses liées au WCET (donc les analyses du cache et des bornes de boucles) de manière tout à fait classique sur le programme principal en modélisant l'appel au composant donné dans la figure 7.4 par la fonction *transfert* donné dans la figure 7.5(b). Considérons par exemple qu'après exécution de ces analyses, les ACS du cache indiquent que, pour le point du programme avant l'appel au composant, $ACS^{avant}_{must}(l_0, 0) = 1$ (pour la simplicité de l'exemple, nous supposons que les blocs de cache à l'extérieur et à l'intérieur du composant suivent le même schéma de numérotation, et nous ne nous préoccupons que d'une ligne de cache, la ligne l_0). Considérons également que $\forall b \neq 0$, $ACS^{avant}_{must}(l_0, b) = 4$, et que l'*AbstractStore* avant le composant indique que $y = 10$. L'application de la fonction résumé fournit les résultats suivants :

- $p_0 = AH$, car $ACS^{avant}_{must}(l_0, 0) = 0$
- $p_1 = NC$ et $p_2 = NC$ car $ACS^{avant}_{must}(l_0, 1) = 4$ et $ACS^{avant}_{must}(l_0, 2) = 4$
- $p_3 = 3 \times 10 = 30$ car l'*AbstractStore* indique que $y = 10$.

Les contraintes paramétrées du système ILP sont donc instanciées comme suit :

- $x^{miss}_0 = 0$
- $e_{1,1} = 30 \times e_{3,1}$

Les contraintes structurelles suivantes sont également présentes :

- $x_3 = e_{3,0} + e_{3,1}$
- $x_1 = e_{3,1} + e_{1,1}$
- $x_1 = e_{1,2} + e_{1,1}$

```
<component name="comp"> <function name="fonction"> <analysis type="instcache">
    <transfer>
      <damage block="0" aging="2" /> <damage block="1" aging="2" >
      <insert block="2" age="0" />
    </transfer>
    <summary> <lblock cacheblockid="1" cond_category="..." /> ... </summary>
  </analysis> <system entry="x1">
    <objective type="max" const="0">
      <term var="x1" coef="12" /> <term var="xmiss_1" coef="32" /> ...
    </objective>
    <constraint op="EQ" const="1"> <term var="x1" coef="1"/> </constraint>
    <switch param="p0">
      <case value="always_hit">
        <constraint op="EQ" const="0" > <term var="xmiss_0" coef="1"/> </constraint>
      </case>
    </switch> <constraint op="LE" const="0" >
       <term var="e_1_1" coef="1"/> <term var="e_3_1" param="p3" coef="-1"/>
    </constraint> ... </system> </function> </component>
```

FIGURE 7.6: Exemple de format XML

- $x_0 = e_{3,0}$
- $x_0 = e_{0,2}$
- $x_2 = e_{0,2} + e_{1,2}$

Ces contraintes-là représentent les aspects du WCET qui sont dépendants du contexte d'entrée du composant, elles sont ajoutées à toutes les contraintes pré-existantes qui ne dépendent pas de l'état d'entrée (comme les contraintes structurelles, les contraintes représentant des catégories de cache ou des limites de boucles invariantes quel que soit l'état d'entrée), ce qui permet de calculer le WCET du programme complet.

A titre d'exemple, nous donnons dans la figure 7.6 une partie du fichier XML qui aurait pu être généré par l'analyse partielle du composant dont le code source est donné dans la figure 7.4. Est représenté le contenu de la fonction *transfert* avec le vieillissement et les blocs insérés, une partie de la fonction *résumé*, ainsi qu'une partie du système ILP associé au composant. Il est possible de voir en particulier la fonction objectif, une contrainte conditionnelle concernant le l-bloc associé au paramètre p_0, ainsi qu'une contrainte paramétrée concernant la borne de boucle associée au paramètre p_3.

7.2 Résolution paramétrique de systèmes ILP

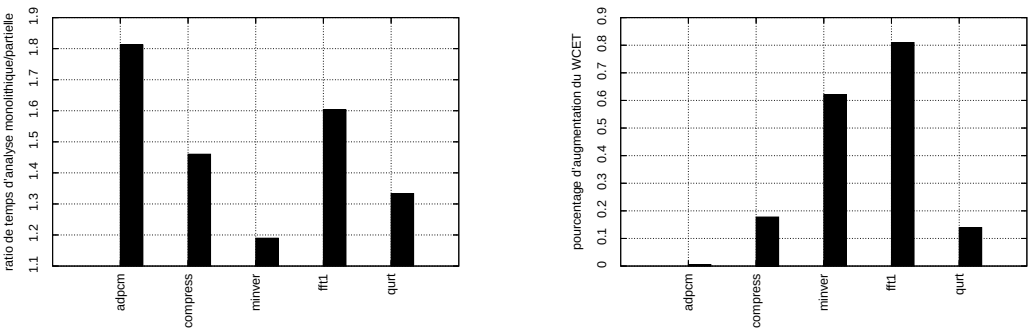
Dans notre approche, le résultat partiel contient un système ILP paramétrique, qui doit ensuite être résolu lors de la composition. Si on peut se permettre de perdre un peu de temps lors de l'analyse partielle, il serait intéressant de pouvoir résoudre ce système paramétriquement, afin de gagner beaucoup de temps lors de la composition. Dans l'article [31], P. Feautrier présente une méthode de calcul ILP paramétrique. Cette méthode permet de résoudre un système ILP dans lequel les contraintes et la fonction objectif peuvent contenir des paramètres. Pour faire un tel système, certaines variables ILP sont désignées comme paramètres (ceci implique, entre autres, qu'on ne peut pas avoir un terme qui est un produit d'une variable ILP par un paramètre, car ceci reviendrait à avoir un produit de deux variables ce qui est interdit dans ILP), ces paramètres sont à valeurs entières. Pour ce faire, l'idée générale est en fait d'appliquer une version symbo-

lique de la méthode de résolution ILP du « *Dual simplex* », la méthode de la coupe de Gomory [37] étant utilisée pour gérer les systèmes ILP entiers. Tous les détails de l'algorithme sont donnés dans [31]. La résolution du système ILP produit non pas un entier, mais un résultat qui dépend des paramètres. Ce résultat paramétré se présente sous forme d'un arbre : chaque nœud représente un test (une inéquation linéaire faisant intervenir un sous-ensemble des paramètres), et chaque feuille représente un résultat final, sous la forme d'une combinaison linéaire des paramètres et/ou une constante. Pour obtenir le résultat concret à partir des valeurs des paramètres et du résultat paramétré, on commence à la racine de l'arbre du résultat, et à chaque nœud le test permet de choisir le prochain nœud fils, en fonction de la valeur des paramètres. Une fois qu'on atteint une feuille, on peut évaluer l'expression du résultat à partir de la valeur des paramètres.

Ce procédé de résolution ILP paramétrique est notamment utilisé par S. Bygde et L. Björn [10, 58], article déjà décrit dans la section 2.3.1. Nous rappelons que dans cet article, le programme et l'ensemble de ses blocs de base sont respectivement notés P et Q_P, tandis que V_P représente l'ensemble des variables. L'ensemble de paramètres d'entrée est noté I_P avec $I_P \subset V_P$. La fonction $MEC_P : \mathbb{Z}^{|I_P|} \rightarrow \mathbb{N}^{|Q_P|}$ permet de calculer les bornes supérieures du nombre de passages par les blocs de base de Q_P (nous noterons $borne_p$ la borne du nombre de passages par le bloc de base $p \in Q_P$), en fonction des paramètres d'entrée I_P. Ces valeurs retournées par la fonction MEC_P constituent les valeurs des paramètres du système ILP. Ainsi, le système ILP construit lors du calcul du WCET est paramétré par le nombre de passages aux points Q_P du programme, c'est-à-dire qu'une contrainte $x_p \leq borne_p$ est générée pour chaque bloc de base p possédant une borne. Lorsque ce système est résolu grâce à la méthode de P. Feautrier, on obtient un résultat paramétrique. Une fois que les paramètres d'entrée de fonction I_P sont connus, la fonction MEC_P est utilisée pour renvoyer les valeurs des paramètres du système ILP, ce qui permet, avec le résultat paramétrique, d'obtenir un WCET concret.

Même si les valeurs $borne_p$ fournissent des informations sur le nombre d'exécutions des blocs de base, on a quand même besoin d'un système ILP : en effet, on ne peut pas simplement trouver le WCET en faisant la somme $\sum_{p \in Q_P} borne_p \times t_p$ (où t_p représente le temps associé au bloc de base p), car les valeurs $borne_p$ sont des bornes supérieures, on a $x_p \leq borne_p$ mais pas forcément $x_p = borne_p$, donc cette méthode entraînerait probablement une grosse sur-estimation du WCET.

A l'avenir, nous pourrions utiliser ce procédé de résolution ILP paramétrique, mais il faudrait toutefois adapter certaines analyses, à cause de l'interdiction d'avoir des termes de la forme $p \times v$, où v est une variable ILP et p est un paramètre. Nous n'avons pas mis en oeuvre cette méthode car pour le moment, avec les moyens matériels et logiciels dont nous disposons, la complexité de cette approche ne permet pas de résoudre en un temps raisonnable des systèmes ILP paramétriques correspondant à des composants issus de benchmarks de taille significative.



(a) Temps de calcul (b) Augmentation du WCET

FIGURE 7.7: Mesures expérimentales

7.3 Expérimentation

Pour valider notre approche, nous l'avons expérimentée avec OTAWA. Nous avons utilisé l'analyse partielle de cache présentée dans le chapitre 4, ainsi que les analyses partielles des limites de boucles et de la prédiction de branchements présentés dans le chapitre 5. Nous avons utilisé le procédé de minimisation du système ILP inclus dans le résultat partiel grâce au pré-calcul des régions indépendantes du contexte. Nous avons utilisé l'architecture cible habituelle, avec les effets de pipeline traités par la méthode des graphes d'exécution.

Les programmes de tests ont été tirés des benchmarks Malärdalen, plus spécifiquement nous avons sélectionnés les benchs *adpcm, compress, minver, fft1,* et *qurt* car ils possèdent une taille suffisante, et il est possible d'y délimiter des composants (principalement des fonctions assez importantes et appelées plusieurs fois dans le programme).

Il est intéressant de voir que l'analyse partielle des composants, en plus de permettre l'usage de COTS, permet une analyse plus rapide car (1) des morceaux de résultats partiels peuvent être réutilisés lorsqu'un point d'entrée de composant est appelé plusieurs fois (réutilisation due à l'algorithme de minimisation grâce aux régions) et (2) le temps de calcul du WCET croît généralement de manière non linéaire, donc il est plus rapide d'analyser des morceaux séparément que d'analyser le tout de manière monolithique. Nous avons mesuré ce gain de temps, les résultats sont dans la figure 7.7(a) : pour chaque test, nous avons montré le ratio de temps entre l'analyse partielle et l'analyse monolithique. Le temps mesuré inclut tout le calcul du WCET, y compris la résolution ILP. Dans le cas de l'analyse partielle, le temps mesuré inclut l'analyse partielle et la composition. En moyenne, l'analyse partielle permet d'aller environ 1.5 fois plus vite, bien que notre composant ne soit appelé que peu de fois dans la plupart des benchmarks. Le gain de temps est plus important lorsque le test utilisé est de grande taille (*adpcm*) car cela permet de pré-évaluer davantage de régions, ou bien que le composant est appelé plusieurs fois *(fft1)* car cela maximise la réutilisation du résultat partiel. Nos tests considèrent uniquement le cas d'un seul programme qui utilise le composant, mais l'intérêt du gain de temps lié à

l'analyse partielle est encore plus évident dans le cas où le résultat partiel associé à un composant est distribué avec celui-ci pour être utilisé un grand nombre de fois, comme présenté dans la section 7.1.1.

Une limitation connue de notre méthode est que les fonctions *transfert* et *résumé* ne sont pas garanties exactes. Elles doivent être toutefois au moins conservatrices (par exemple, pour le *transfert* du cache, la valeur $ACS_{must}^{après}$ qui aurait été calculée par une vraie analyse du composant doit être incluse dans celle calculée par $transfert(ACS_{must}^{avant})$. Ceci peut aboutir à du pessimisme, que l'on a mesuré : dans la figure 7.7(b), pour chaque test on mesure le pourcentage d'augmentation du WCET calculé par analyse partielle, par rapport à celui calculé par analyse monolithique. Heureusement, on remarque qu'en moyenne, le WCET n'est augmenté que de 0.4 % (le pessimisme est causé uniquement par l'approximation des fonctions *transfert* et *résumé* car le procédé de minimisation du système grâce aux régions est exact).

Dans ce chapitre, nous avons proposé une approche générique pour traiter les composants "boîte noire" dans le calcul du WCET. Ceci nous permet de calculer un résultat partiel pour des composants, et ensuite d'utiliser ces résultats partiels lors du calcul du WCET de la tâche utilisant les composants. En étudiant les diverses analyses partielles existantes basées sur IPET (le cache d'instructions, l'estimation des bornes de boucles), nous avons pu proposer une approche générale qui pourra être facilement extensible à d'autre analyses.

Bien que cette approche soit principalement conçue pour la gestion des composants "boîte noire", les tests réalisés avec OTAWA ainsi qu'avec oRange ont montré que (1) à cause de la croissance non linéaire du temps de calcul WCET et (2) grâce à la réutilisation des résultats partiels, il y a un gain de temps non négligeable lorsqu'on utilise l'analyse partielle.

Chapitre 8

Réalisation

Dans ce chapitre, nous exposons la réalisation technique des analyses que nous avons conçues. Ces analyses ont été pour la plupart implémentées grâce à OTAWA, notre outil de calcul de WCET (d'autres outils de WCET ont également été développés par différentes équipes [48, 45, 85]). C'est pourquoi nous commençons par une présentation d'OTAWA, avant d'exposer le développement qui a été effectué dans cet outil afin d'implémenter les méthodes développées.

8.1 OTAWA

Pour toutes nos expérimentations, nous utilisons OTAWA, l'outil d'analyse et de calcul de WCET réalisé dans mon équipe [11]. Les besoins pour faire une analyse de WCET évoluent rapidement, en fonction notamment de l'architecture cible. C'est pourquoi, bien qu'OTAWA contienne des outils permettant directement de calculer le WCET sur un bon nombre d'architectures cibles, OTAWA est avant tout un *framework* permettant de faciliter l'écriture de nouvelles analyses, de gérer de nouvelles architectures cibles ou de nouvelles méthodes de calcul.

8.1.1 Présentation générale d'OTAWA

Puisque OTAWA doit être adaptable à de nouvelles architecture et méthodes, il possède une architecture modulaire, lui permettant de s'adapter à diverses architectures et techniques de calcul de WCET. Un système de plug-ins permet de placer la plupart des fonctionnalités d'OTAWA dans des modules externes, qui seront localisés et chargés si nécessaire. OTAWA est composé d'un ensemble d'analyseurs (appelés processeurs de code) pouvant réaliser des tâches diverses, de la construction du CFG à partir du binaire jusqu'au calcul final du WCET. Ces processeurs, qui peuvent être dans le coeur d'OTAWA ou bien dans des plug-ins, mettent à jour un *workspace* (espace de travail) qui contient le programme analysé ainsi que les informations liées aux calcul du WCET (données d'entrées, résultats intermédiaires et résultats finaux). Chaque processeur peut donc exploiter les informations stockées dans le *workspace* par un processeur précédent, et lui-même sto-

cker des informations utiles pour les processeurs suivants. Ces informations sont stockées grâce à des annotations (appelées propriétés), qui peuvent être accrochées à de nombreux types d'éléments du programme (CFG, bloc de base, instructions, ...). Ainsi, par exemple un processeur chargé de déterminer les bornes de boucles pourra utiliser les propriétés accrochées aux nœuds de boucles permettant d'identifier celles-ci, et de stocker sur ces même nœuds de boucle le nombre d'itérations maximum.

Comme les processeurs peuvent avoir besoin des informations calculées par d'autre processeurs, un système de dépendance entre les processeurs de code permet à OTAWA d'enchaîner les analyses dans le bon ordre pour aboutir au WCET. Pour ce faire, le concept de *feature* a été introduit. Une *feature* représente un des résultats d'un processeur et est associé à un ensemble de propriétés. La façon de regrouper les propriétés est arbitraire, mais en général pour des raisons de simplicité on regroupe dans une *feature* les propriétés qui sont souvent utilisées ensemble par les processeurs. Chaque processeur peut dépendre d'un ensemble de *features,* et lui-même produire un autre ensemble de *features* comme résultat. Plusieurs processeurs peuvent produire la même *feature*, mais une *feature* désigne en général un processeur « par défaut » chargé de la produire.

Par exemple, la *feature* appelée « LOOP_INFO_FEATURE » dans OTAWA est associée aux propriétés LOOP_HEADER, EXIT_EDGE, et ENCLOSING_LOOP qui expriment respectivement les informations sur les têtes de boucles, les arcs de sortie, et les relations d'inclusion entre différentes boucles. Le processeur « LoopInfoBuilder » qui fait l'analyse des boucles produit la *feature* « LOOP_INFO_FEATURE ». Le processeur qui se charge de fournir les bornes de boucles contient « LOOP_INFO_FEATURE » (entre autres) dans sa liste de dépendances. Ce système permet de s'assurer qu'un processeur ne sera pas lancé tant que le *workspace* ne contient pas toute les informations nécessaires à son fonctionnement. De plus, comme chaque *feature* est associée à un processeur par défaut qui la produit, lorsque OTAWA tente de lancer un processeur, mais qu'il dépend d'une *feature* non présente dans le *workspace* en cours, OTAWA peut automatiquement lancer au préalable le processeur par défaut associé à la *feature* manquante.

La figure 8.1 montre un exemple d'ensemble de processeurs et de features, depuis la construction du CFG (processeur *CFGBuilder*) jusqu'au calcul final du WCET (processeur *WCETComp*). Le système de dépendances assure que les processeurs *ConsBuilder*, *FlowFactLoader* et *ObjectBuilder* (qui sont tous nécessaires pour la construction du système ILP) seront appelés. Les processeurs *ExeGraph* et *BBSimul* fournissent tous deux la feature *BBTime* qui représente la disponibilité du temps associé à chaque bloc de base (*ExeGraph* emploie la méthode des graphes d'exécution, tandis que *BBSimul* simule l'exécution de chaque bloc de base à l'aide d'un simulateur fonctionnel pour obtenir son temps d'exécution). La feature *BBTime* doit être associée à un processeur par défaut capable de la produire (*ExeGraph* ou bien *BBSimul*, ou même un autre processeur non listé ici) .

Ceci permet donc, dans le plus simple des cas, de lancer un processeur permettant de calculer le WCET, et grâce aux systèmes de dépendances, OTAWA enchaînera les analyses

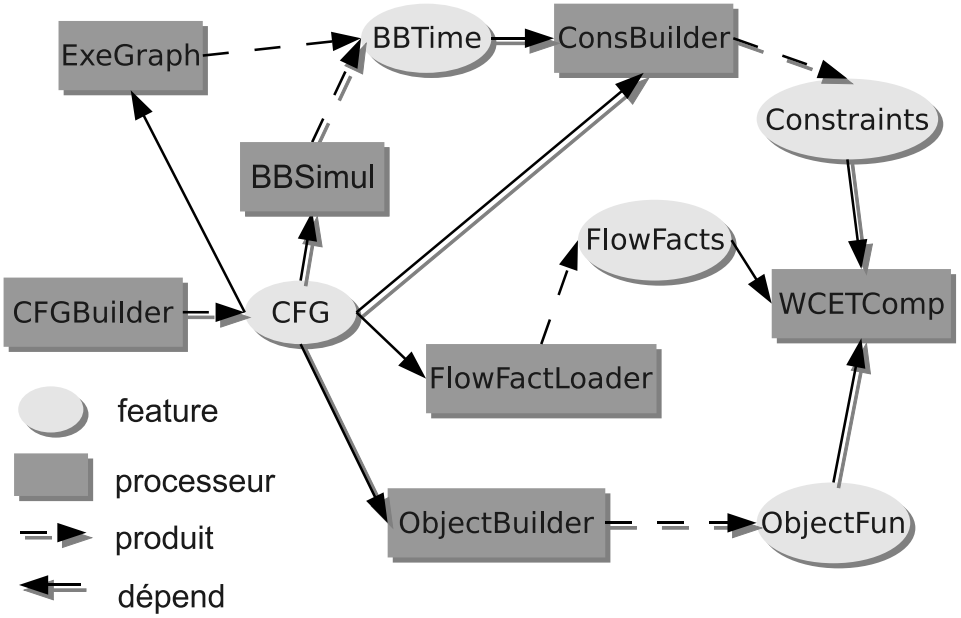


FIGURE 8.1: Les features : exemple

requises dans le bon ordre. Bien sûr, les processeurs ainsi lancés automatiquement seront des processeurs par défaut, avec la configuration par défaut, donc ceci ne produira pas forcément le WCET le plus précis qu'il soit possible d'obtenir. C'est pourquoi il est aussi possible de lancer des processeurs manuellement, et dans ce cas il est possible de spécifier un ensemble de propriétés de configuration, qui peuvent altérer le fonctionnement du processeur.

8.1.2 Processeurs de code dans OTAWA

Il y a des processeurs pour réaliser toutes les tâches utiles dans OTAWA pour le calcul de WCET. Ceci inclut notamment :

- Tous les processeurs « bas niveau » qui sont chargés de reconstituer l'image du programme à analyser à partir du fichier binaire. Ceci inclut les *loaders* pour un certain nombre d'architectures et de format de fichiers binaires. Le système de plug-in d'OTAWA permet de trouver le *loader* correspondant au fichier binaire qu'on souhaite analyser. Les *loaders* pour la plupart des architectures sont générés automatiquement à partir d'une description du processeur par l'outil GLISS. Cet outil (externe à OTAWA) prend en entrée une description d'un processeur (en SimNML [8]) et génère le code en C d'un simulateur, désassembleur, et débogueur. Tout ceci permet à cette partie « bas niveau » d'OTAWA de fournir une abstraction du programme à analyser, indépendante de l'architecture. Par exemple, il est possible d'avoir des informations sur la sémantique des instructions du programme quel que soit le jeu d'instruction de l'architecture (type d'opération, branchement conditionnel/inconditionnel, lecture/écriture de registres et/ou mémoire, etc). Les autres processeurs, qui travaillent sur une représentation de plus haut niveau, peuvent se baser sur cette abstraction.

- Les processeurs qui travaillent sur la représentation du programme. Ceci inclut les processeurs chargés d'identifier les diverses structures du programme (blocs de base, fonctions, boucles, etc.) de construire le CFG, le PCG (Procedure Call Graph), le PST (Program Structure Tree), le Context Tree, les processeurs chargés de faire le déroulement de boucles et l'inlining de fonctions, etc.
- Les processeurs liés aux diverses analyses de flot ou d'effets architecturaux nécessaires pour le WCET (ceci représente la majorité des processeurs d'OTAWA), comme le chargement des bornes de boucles, ou bien l'analyse du cache et de la prédiction de branchement. Ces processeurs peuvent être liés à une technique spécifique de calcul de WCET (IPET, ETS, etc) ou bien être indépendants.
- Les processeurs de calcul de WCET, liés à une technique spécifique. Par exemple pour IPET, il y a des processeurs pour générer le système ILP de base (contraintes structurelles, fonction objectif, etc), pour lancer la résolution du système ILP, etc. La résolution du système ILP est implémentée par un plug-in, ce qui permet d'utiliser plusieurs moteurs de résolution ILP. Actuellement, OTAWA utilise par défaut *lp_solve*[60], un programme de résolution ILP open source. Une fonction d'export a été également écrite pour le système de résolution ILP paramétrique de P. Feautrier (utilisé dans l'article [10]), PIPlib.

8.1.3 HalfAbsInt

Dans le calcul de WCET, un bon nombre d'analyses utilisent de l'interprétation abstraite. Nous pouvons citer les exemples de l'analyse de cache et de prédiction de branchement, étudiés dans le présent document. Il peut être utile, comme présenté dans [2], de créer un outil permettant de développer des analyses par interprétation abstraite. C'est pourquoi nous avons développé *HalfAbsInt*, un moteur générique d'interprétation abstraite intégré à OTAWA. Ce module a été utilisé dans beaucoup de processeurs d'OTAWA.

HalfAbsInt gère l'interprétation abstraite appliquée aux CFGs, tel que décrit dans la section 2.1.6, et produit comme résultat un état abstrait avant et après chaque bloc de base du CFG. HalfAbsInt est implémentée sous forme d'un *template* qui est paramétré par divers éléments : le paramètre `Problem`, le paramètre `FixPoint`, et le paramètre `Listener`. La partie principale de HalfAbsInt s'occupe du parcours du CFG, et utilise le `FixPoint` pour gérer les boucles. Le `Problem` est utilisé pour définir le domaine de l'interprétation abstraite ainsi que la façon dont il est mis à jour lors du parcours du CFG, tandis que le `Listener` est utilisé pour collecter les résultats.

Tout d'abord, décrivons le paramètre `Problem` de manière un peu plus détaillée. Une interprétation abstraite sur un CFG est définie par un certain nombre de paramètres (domaines, fonctions d'update et de join, etc), ces paramètres sont stockés dans la classe qui correspond au paramètre `Problem`. Une instance de cette classe est donc fournie par l'utilisateur, et définit donc un problème particulier d'interprétation abstraite. Un objet de cette classe contiendra donc tous les attributs permettant de définir le problème :

- Le domaine abstrait (type `Domain`), représentant l'ensemble de toutes les valeurs

abstraites possibles, muni de son $\top$ (plus grand élément), de son $\bot$ (plus petit élément), et de son opération *lub* (plus petite borne supérieure entre deux éléments). La classe `Domain` contient également un opérateur d'égalité pour déterminer si deux états abstraits sont égaux (indispensable pour savoir lorsqu'on a atteint le point fixe dans une boucle), ainsi qu'un opérateur d'assignation.
- L'état d'entrée à utiliser lors de l'analyse (le plus souvent $\top$, mais ceci est paramétrable par l'utilisateur).
- La méthode *update* qui associe à chaque bloc de base une mise à jour de l'état abstrait. Cette fonction prend donc en paramètre un bloc de base et un élément de `Domain`, et renvoie l'élément `Domain` après mise à jour.
- La méthode *join* qui sert à fusionner les états abstraits en cas de jonction de chemins dans le CFG. Généralement la fonction *join* est équivalente à l'opération *lub* de `Domain`.
- Les méthodes *entercontext()* et *leavecontext()* prennent en paramètre un identifiant de contexte ainsi qu'un état abstrait à modifier. Ces fonctions, qui peuvent éventuellement être vides, permettent de réaliser une opération sur l'état abstrait lorsqu'on entre ou sort d'un contexte (un contexte, dans ce cas, est une boucle ou une fonction). Ceci est nécessaire, par exemple, pour gérer la *persistence* paramétrique : *entercontext()* et *leavecontext()* sont utilisées pour empiler ou dépiler les éléments de la *persistence* paramétrique lorsqu'on rentre dans une boucle et qu'on la quitte.

A partir de cette classe `Problem`, HalfAbsInt est capable de réaliser l'interprétation abstraite sur un CFG sans boucles, comme indiqué dans la première partie de la section 2.1.6, en appliquant les fonctions *update* et *Join* à partir de l'état d'entrée. Pour ce faire, HalfAbsInt applique l'algorithme présenté dans la figure 2.4.

Toutefois, ceci ne permet pas de gérer les boucles. Pour ce faire, on a besoin de les identifier, et d'avoir un mécanisme qui permet d'itérer le processus d'interprétation abstraite sur ces boucles jusqu'à converger vers un point fixe.

L'identification des boucles (c'est-à-dire la détection de leur tête de boucle, des arcs de retour, d'entrée, et de sortie, des blocs de base appartenant à une boucle, ainsi que la détection des relations d'inclusion entre différentes boucles) est réalisée par `LoopInfoBuilder`, un processeur d'OTAWA qui doit impérativement être exécuté avant toute utilisation d'HalfAbsInt. La gestion des boucles et de la convergence vers le point fixe est réalisée par la classe qui implémente le paramètre template `FixPoint`. Pour ce faire, cette classe doit posséder une méthode `fixPoint`() qui est appelée par HalfAbsInt à chaque fois qu'une nouvelle itération d'une boucle débute. Cette méthode prend en paramètre (1) la boucle (le numéro du bloc de base tête de boucle), (2) l'état abstrait résultant de l'union des arcs d'entrée de la boucle, ainsi que (3) l'état abstrait résultat de l'union des états abstraits des arcs de retour. Elle met à jour l'état abstrait associé à la tête de boucle, et retourne un booléen qui indique à HalfAbsInt s'il faut continuer à itérer sur cette boucle. Cette méthode est appelée par HalfAbsInt chaque fois qu'on passe par une tête de boucle. C'est donc la classe qui implémente `FixPoint` qui décide comment évolue l'état abstrait associé

à la tête de boucle, et qui détermine lorsque le point fixe est atteint. Une classe par défaut, `DefaultFixPoint`, est fournie dans l'implémentation, mais l'utilisateur peut aussi fournir une version personnalisée. La classe par défaut applique l'algorithme présenté dans la section 2.1.6 (mise à jour simple de l'état de tête de boucle grâce aux états des arcs d'entrée et des arcs retour, avec arrêt lorsque l'état de tête de boucle ne varie plus), sans utiliser d'opérateur de *widening* (utilisation du *join* de `Problem` pour gérer les jonctions).

En présence d'un CFG contenant des boucles, le moteur d'HalfAbsInt applique donc l'algorithme de la figure 2.4 avec la modification suivante : lorsqu'une tête de boucle est rencontrée (la structure du CFG et de l'algorithme de parcours implique qu'on ne peut pas rencontrer un bloc appartenant à une boucle sans avoir au préalable rencontré la tête de boucle), le corps de la boucle est interprété itérativement, en utilisant le `FixPoint` à chaque itération jusqu'à ce qu'il indique que l'évaluation de la boucle est terminée. Alors, les arcs de sortie de boucles sont annotés avec les états abstraits qui résultent de l'évaluation de la boucle, et l'interprétation de l'extérieur de la boucle peut continuer.

La classe par défaut, `DefaultFixPoint`, permet de traiter la majorité des cas ; toutefois une autre classe est fournie, `UnrollingFixPoint`, permettant de gérer le déroulement virtuel de boucles (l'interprétation abstraite donne les résultats qui auraient été obtenus sur un CFG à boucles déroulées, sans qu'il soit nécessaire de réaliser ce déroulage). Cette flexibilité ouvre la porte a diverses possibilités, comme l'écriture d'une implémentation de `FixPoint` qui utilise un opérateur de *widening*.

Le dernier paramètre template, `Listener`, permet de récupérer les résultats de l'interprétation abstraite, c'est-à-dire les états abstraits aux endroits du programme qui nous intéressent. Pour ce faire, le `Listener` contient un ensemble de méthodes qui sont appelées par HalfAbsInt lors des différentes étapes de l'interprétation abstraite, c'est-à-dire au fur et à mesure que des résultats (intermédiaires ou définitifs) sont obtenus. Par l'intermédiaire de ces appels de méthodes, HalfAbsInt passe au `Listener` des informations (états abstraits) sur l'analyse, que la méthode du `Listener` peut stocker comme elle le souhaite, ou ignorer. Les méthodes dont dispose le `Listener` sont les suivantes :

- la méthode `blockInterpreted()` est appelée à chaque fois que `HalfAbsInt` a traité un bloc de base. Cette méthode permet de fournir au `Listener` l'état abstrait avant et après ce bloc de base, ainsi que le numéro d'itération des boucles qui contiennent le bloc de base.
- la méthode `fixPointReached()` est appelée chaque fois que le point fixe a été atteint au niveau d'une boucle (et donc lorsqu'on est sur le point de quitter cette boucle). Cette méthode permet de passer au `Listener` l'état de tête de boucle final.

Il existe une classe par défaut appelée `DefaultListener`, qui implémente `Listener`. Dans cette classe, uniquement la méthode `blockInterpreted()` est utilisée. Cette implémentation par défaut enregistre, pour chaque endroit du programme (avant et après chaque bloc de base), l'état abstrait correspondant à l'union de tous les états abstraits à ce point. La classe DefaultListener fournit aussi des méthodes permettant d'accéder à ces résultats.

De plus, la classe `UnrollingListener` est aussi fournie comme implémentation du

`Listener` dans OTAWA. Cette classe est prévue pour être utilisée avec `UnrollingFixPoint`, et permet donc le déroulement virtuel des boucles. Dans ce cas, les résultats pour chaque bloc de base sont répertoriés distinctement pour chaque contexte de boucle, selon le numéro d'itération.

Chapitre 9

Conclusion

9.1 Résultats

Ce travail a eu pour but le développement d'une méthode d'analyse partielle pour le calcul de WCET par analyse statique. Une telle méthode doit permettre d'analyser un composant logiciel pour donner un résultat partiel, qui doit pouvoir ensuite être utilisé pour représenter l'appel au composant lors du calcul de WCET du programme principal. Les intérêts de l'analyse partielle sont (1) la possibilité de gérer les composants « boîte noire » dans l'analyse de WCET, et (2) le gain de temps d'analyse, rendu possible par la division du travail à accomplir en plusieurs parties.

Pour faire ce travail, nous avons étudié un certain nombre d'analyses pré-existantes, et nous avons adapté certaines d'entre elles, afin d'obtenir un fondement solide permettant de s'attaquer au problème de l'analyse partielle. La plus importante de ces analyses est celle de la prédiction du cache d'instructions.

Nous avons donc adapté cette analyse pour qu'elle soit plus propice à l'analyse partielle des composants. En particulier, la *persistence* paramétrique permet de localiser la portée des blocs *persistents*, et donc favorise la détection des effets locaux à un composant donné.

En plus de favoriser l'analyse partielle, cette *persistence* paramétrique a aussi permis de se passer du déroulement de boucles (car les effets qui étaient pris en charge par le déroulement de boucles sont maintenant détectés par la *persistence*), ce qui rend l'analyse plus rapide à cause de la réduction du nombre de variables et contraintes ILP. De plus, l'analyse de cache avec déroulement de boucles ne permettait pas de prendre en charge tous les effets de *persistence* possibles, donc la *persistence* paramétrique permet aussi de gagner de la précision (WCET moins pessimiste). Les mesures effectuées grâce à OTAWA ont permis de mettre en évidence ce gain de temps et de précision.

A partir de l'expérience acquise lors du développement de ces analyses de cache d'instructions, nous avons pu mettre en place d'autres analyses pour des mécanismes matériels au comportement similaire (comme le *row buffer*). Les résultats montrent que ces analyses fonctionnent, et pourront facilement être converties en analyse partielle.

A partir de l'analyse de cache réalisée pour le cache d'instructions, nous avons développé l'analyse partielle correspondante. Cette analyse partielle permet, à partir d'un

composant, d'obtenir un résultat partiel composé de deux parties principales. Premièrement la fonction *dommage* permet de modéliser l'effet qu'a l'appel au composant sur l'état abstrait de l'analyse de cache, en déterminant quels blocs de cache sont vieillis ou insérés dans le cache lors de l'exécution du composant. Deuxièmement, la fonction *résumé* donne une version paramétrique des catégories de cache des blocs du composant, qui peut être instanciée lorsqu'on doit calculer le WCET du programme principal.

Cette analyse partielle prend en compte tout ce qui est géré par l'analyse de cache d'instructions que nous avons développée, y compris la *persistence* paramétrique. Les résultats que nous avons obtenus grâce à OTAWA (sur un sous-ensemble des tests Malärdalen adaptés à l'analyse partielle) montrent que cette analyse partielle fonctionne tout en ajoutant très peu de pessimisme au WCET, et qu'il y a un gain de temps significatif, surtout dès que le composant à traiter est appelé plusieurs fois (ce qui permet de faire de la réutilisation de résultats).

Pour ce qui est de la prédiction des bornes de boucles, nous avons passé quelques analyses existantes en revue, et l'analyse réalisée par Z. Ammarguellat et al. [4] et plus tard améliorée par M. de Michiel et al. [24] nous a paru une des mieux adaptées à l'analyse partielle. En effet, cette analyse était déjà munie d'une interprétation abstraite dont les éléments du domaine (les *AbstractStore*) possédaient déjà une opération de composition, et les résultats étaient d'abord exprimés de manière symbolique avant d'être calculés numériquement. A partir de cette analyse, nous avons fait des adaptations qui nous ont permis de développer l'analyse partielle des bornes de boucles. Cette analyse partielle permet d'obtenir, à partir d'un composant, un résultat partiel qui permet d'une part de prendre en compte l'influence du composant sur l'état global du programme (variables globales ou statiques, paramètres passés par référence, etc.), mais aussi d'exprimer les bornes de boucles du composant sous forme paramétrique (ce qui permet de les instancier pour chaque contexte d'appel lors de l'analyse du programme appelant). Les résultats qui ont été obtenus grâce à l'expérimentation à l'aide d'oRange nous ont permis de constater que cette analyse partielle donnait des résultats satisfaisants, autant en terme de précision qu'en terme de rapidité, et les rares cas qui causaient de la perte de précision sur les bornes de boucles ont été identifiés.

Nous avons développé une analyse partielle de la prédiction de branchement en nous basant sur la méthode de catégorisation présentée par I. Puaut et al. [15]. Nous avons choisi de nous baser sur cette approche car nous avions montré que des approches similaires (basées sur le principe de catégorisation des caches) se prêtaient facilement à l'analyse partielle ; de plus cette méthode offre un bon compromis entre la rapidité d'exécution et la précision. Toutefois, la méthode de I. Puaut et al. ayant été développée pour l'approche ETS (Extended Timing Schema), nous avons dû l'adapter à l'approche IPET que nous utilisons. Nous avons donc implémenté notre propre version de l'analyse de prédiction de branchement par catégorisation compatible avec l'approche IPET. Notre méthode permet aussi de gérer certains certains effets du prédicteur de branchements qui n'étaient pas pris en compte par la méthode de I. Puaut et al. Nous avons testé cette technique grâce à

OTAWA, les résultats sont satisfaisants, avec plus de 90% des instructions de branchement catégorisées correctement, et le temps d'analyse est rapide. En partant de cette analyse, nous avons proposé une méthode d'analyse partielle du prédicteur de branchements. Pour ce faire, nous avons procédé de manière similaire à ce que nous avions fait pour l'analyse partielle du cache d'instructions : nous avons modélisé l'effet du composant sur le reste du programme par une fonction *dommage* qui prend en compte les instructions de branchement qui sont vieillies ou insérées par l'exécution du composant, et nous avons développé une fonction *résumé* qui joue le même rôle que la fonction *résumé* du cache, avec les catégories des instructions de branchement.

Pour répondre au problème du temps de calcul important de la résolution ILP, nous avons proposé notre approche de découpage du CFG en régions, en nous basant sur l'algorithme de construction du *Program Structure Tree* décrit par R. Johnson et al. [50]. Nous avons montré comment découper le CFG à analyser en régions de telle manière que le WCET obtenu reste le même. Pour ce faire, nous avons donné une méthode pour identifier les régions de CFG qui sont indépendantes (c'est-à-dire dont le WCET ne dépend pas du contexte environnant). Pour ce faire, nous avons développé un moyen générique permettant aux diverses analyses impliquées dans le WCET (comme le cache, la prédiction de branchement, ou les bornes de boucles) de fournir des informations permettant à notre algorithme de déterminer si une région est indépendante. Le résultat de ce programme est un PST dans lequel tous les nœuds correspondent à des régions dont le WCET est fixe. Les résultats expérimentaux obtenus à l'aide d'OTAWA montrent que le WCET calculé est strictement identique, et que le temps d'analyse est très réduit.

A partir des versions partielles des diverses analyses, ainsi que de la méthode de découpage en régions, nous avons pu proposer une méthode générale de calcul partiel du WCET.

9.2 Perspectives

L'analyse partielle présente principalement deux intérêts, la gestion des composants « boîte noire » type COTS, et la réduction du temps de calcul. La prise en compte des composants a pu être gérée grâce aux méthodes d'analyse partielle que nous avons présentées, et la vitesse du temps de calcul se trouve également augmentée. Toutefois, afin d'utiliser l'analyse partielle exclusivement pour gagner du temps d'analyse, il faudrait avoir un moyen de partitionner automatiquement le programme à analyser en composants (contrairement au cas du COTS, où le partitionnement est imposé). Ceci existe déjà pour la résolution IPET (le découpage du CFG en régions SESE est un partitionnement automatique qui permet de réduire le temps de résolution ILP), mais il serait intéressant de réaliser la même chose pour toutes les analyses qui contribuent au WCET, telles que le cache, la prédiction de branchement ou la détermination des limites de boucle (de plus, le partitionnement peut très bien être différent pour chaque analyse contribuant au WCET, pour faire en sorte qu'on obtienne les meilleures performances possibles).

Le résultat partiel calculé par notre méthode retourne un système ILP paramétré pour chaque composant analysé. Ce système ILP paramétré a été préalablement réduit par le calcul de toutes les régions indépendantes du contexte d'appel du composant. Toutefois, après avoir été instancié grâce aux paramètres correspondant au contexte d'appel, ce système doit être résolu par ILP, et ce pour chaque appel au composant (ou du moins, pour chaque appel au composant comportant une attribution unique des paramètres de contexte), ce qui prend du temps. Pour cette raison, il serait intéressant de voir s'il est faisable de remplacer ce système paramétré par un résultat symbolique, calculé selon la méthode utilisée dans [10]. Pour ce faire, il faudrait adapter certaines analyses partielles qui produisent des termes du type $p_n v_n$ (où p_n est un paramètre, et v_n une variable ILP) car ce genre de termes n'est pas supporté par la méthode de résolution ILP paramétrique (l'approche de S. Bygde utilise l'ILP paramétrique pour prendre en compte principalement des informations de flot de contrôle, alors que notre approche doit, en plus, prendre en compte les effets matériels). Cette approche n'empêcherait pas de continuer à réduire les parties du système ILP ne dépendant pas du contexte, mais cela permettrait en outre de réduire le temps d'instanciation d'un composant dont le résultat ILP serait déjà précalculé.

La méthode que nous avons présentée prend en compte un certain nombre d'analyses que nous avions préalablement sélectionnées (cache, prédiction de branchement, boucles) et est orientée IPET. Il serait intéressant d'explorer les possibilités pour d'autres analyses impliquées dans le WCET. De plus, il serait utile de pouvoir obtenir les fonctions *transfert* (et donc aussi les fonctions *résumé*) à partir d'une description formelle du fonctionnement de l'analyse. En particulier pour les analyses basées sur l'interprétation abstraite, il serait intéressant d'avoir un moyen pour, à partir de la description du *update* et du *join,* pouvoir dériver automatiquement une fonction *transfert* valide pour un composant donné. Pour ce faire, nous avons commencé à travailler sur une méthode qui permettrait, à partir d'une description du problème associé à une interprétation abstraite (fonctions *Update/Join* et domaine) et d'un CFG, de calculer une expression représentant les différentes étapes de calcul nécessaires à l'exécution de cette interprétation abstraite sur ce CFG. Cette expression peut ensuite être automatiquement simplifiée grâce aux propriétés données avec la description du problème, l'expression résultante peut être utilisée pour la construction automatique d'une fonction *transfert*. Ceci permettrait d'adapter plus rapidement et facilement les analyses de WCET à notre méthode d'analyse partielle.

Bibliographie

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of SAS'95, Static Analysis Symposium*, volume LNCS 983, pages 33–50, September 1995.

[3] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96 : Proceedings of the Third International Symposium on Static Analysis*, pages 52–66, London, UK, 1996. Springer-Verlag.

[4] Zahira Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. *SIGPLAN Not.*, 25(6) :283–295, 1990.

[5] C. Ballabriga and H. Cassé. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *Euromicro Conference on Real-Time Systems (ECRTS), Prague, 02/07/08-04/07/08*, July 2008.

[6] C. Ballabriga, H. Cassé, and P. Sainrat. WCET computation on software components by partial static analysis. In *Junior Researcher Workshop on Real-Time Computing, Nancy, 29/03/2007-30/03/2007*, pages 15–18, http ://www.loria.fr, March 2007. LORIA.

[7] C. Ballabriga, H. Cassé, and P. Sainrat. An improved approach for set-associative instruction cache partial analysis. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*, March 2008.

[8] Souvik Basu and Rajat Moona. High Level Synthesis from Sim-nML Processor Models. In *VLSI 2003*, pages 255–260, January 2003.

[9] Roman Bourgade, Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Accurate analysis of memory latencies for WCET estimation. In *International Conference on Real-Time and Network Systems (RTNS), Rennes, 16/10/08-17/10/08*, pages 161–170, http ://www.irisa.fr/, octobre 2008. IRISA.

[10] Stefan Bygde and Björn Lisper. Towards an automatic parametric wcet analysis. In *WCET*, 2008.

[11] H. Cassé and P. Sainrat. OTAWA, a framework for experimenting WCET computations. In *3rd European Congress on Embedded Real-Time Software*, 25-27 December 2005.

[12] C. Cifuentes. A structuring algorithm for decompilation. In *XIX Conferencia Latinoamerica de Informatica*, pages 267–276, aug 1996.

[13] A. Colin and I. Puaut. Worst-case timing analysis of the RTEMS real-time operating system. Technical Report 1277, IRISA, 1999.

[14] A. Colin and I. Puaut. A modular & retargetable framework for tree-based WCET analysis. In *ECRTS*, pages 37–44, 2001.

[15] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3) :249–274, 2000.

[16] Antoine Colin, Isabelle Puaut, Christine Rochange, and Pascal Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art . Rapport de recherche 2002-15-R et PI1461, IRIT et IRISA, Université Paul Sabatier, Toulouse et université Rennes II, mai 2002.

[17] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. Software practice and experience, no. 4. (2001), 2001.

[18] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2) :324–328, June 1996.

[19] P. Cousot. Program analysis : The abstract interpretation perspective. *ACM Computing Surveys*, 28A(4es) :165–es, December 1996.

[20] P. Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1), January 2000.

[21] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[22] 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.

[23] C. Cullman and F. Martin. Data-Flow Based Detection of Loop Bounds. In *7th International Workshop on Worst-Case Execution Time Analysis*, pages 53–58, 3 July 2007.

[24] Marianne de Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Kaohsiung, Taiwan, 25/08/2008-27/08/2008*, pages 161–168, http ://www.computer.org, août 2008. IEEE Computer Society.

[25] J. Engblom. *Processor Pipelines and and Static Worst-Case Execution Time Analysis*. PhD thesis, University of Uppsala, 2002.

[26] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In IEEE Computer Society Press, editor, *Proc. of 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, December 1999.

[27] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation and Invariant Analysis. In *7th International Workshop on Worst-Case Execution Time Analysis*, pages 59–64, 3 July 2007.

[28] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *European Conference on Parallel Processing*, pages 1298–1307, 1997.

[29] Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. Clustered worst-case execution-time calculation. *IEEE Transaction on Computers*, 54(9) :1104–1122, September 2005.

[30] M. Erne, J. Koslowski, A. Melton, and G.E. Strecker. A primer on galois connections. In *York Academy of Science*, 1992.

[31] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Op'erationnelle*, 22, 1988.

[32] C. Ferdinand. A fast and efficient cache persistence analysis. *Technical report, Universitat des Saarlandes*, Sept. 1997.

[33] C. Ferdinand, R. Heckmann, H. Theiling, and R. Wilhelm. Convenient user annotations for a WCET tool. In J. Gustafsson, editor, *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET'03*, volume MDH-MRTC-116/2003-1-SE, pages 17–20. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden, 1 July 2003.

[34] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 37–46, 1997.

[35] C. Ferdinand, J. Schneider, and R. Wilhelm. Pipeline behavior prediction for superscalar processors. Technical report, 31 January 1999.

[36] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. *Lecture notes in computer science*, pages 16–30, 1998.

[37] R. E. Gomory. An algorithm for integer solutions to linear programming. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302, New York, 1963. McGraw-Hill.

[38] Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. A tool for automatic flow analysis of C-programs for WCET calculation. In *WORDS*, pages 106–112. IEEE Computer Society, 2003.

[39] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench : A free, commercially representative embedded benchmark suite. In *WWC '01 : Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[40] Damien Hardy and Isabelle Puaut. Predictable code and data paging for real time systems. In *ECRTS*, pages 266–275, 2008.

[41] Damien Hardy and Isabelle Puaut. Wcet analysis of multi-level set-associative instruction caches. *CoRR*, abs/0807.0993, 2008.

[42] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4) :557–567, 1997.

[43] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 12–21, Washington - Brussels - Tokyo, June 1998. IEEE.

[44] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1) :53–70, January 1999.

[45] R. Heckmann and C. Ferdinand. Verifying Safety-Critical Timing and Memory-Usage Properties of Emebedded Software by Abstract Interpretation. *Proc. of Design Automation and Test in Europe (DATE'05)*, pages 618–619, 2005.

[46] 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*, 91(7) :1038 – 1054, July 2003.

[47] N. Holsti, T. Langbacka, and S. Saarinen. Using a worst-case execution time tool for real-time verification of the debie. In *Software, Proceedings of the DASIA 2000 (Data Systems in Aerospace) Conference (ESA SP-457, ISBN*, pages 92–9092, 2000.

[48] N. Holsti and S. Saarinen. Status of the Bound-T tool. In *2nd International Workshop on Worst-Case Execution Time Analysis (WCET'2002)*, June 2002.

[49] Niklas Holsti. Computing time as a program variable : a way around infeasible paths. In Raimund 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.

[50] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree : Computing control regions in linear time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–185, 1994.

[51] S.K. Kim, S.L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 230–240, 1996.

[52] X. Li, A. Roychoudhury, and T. Mitra. Modeling Out-of-Order Processors for Software Timing Analysis. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 92–103. IEEE Computer Society, 2004.

[53] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 88–98, 1995.

[54] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modelling and path analysis for real-time software. *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 254–263, December 1995.

[55] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software : Beyond Direct Mapped Instruction Caches. *Proceedings of ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-time Systems*, pages 47–55, June 1997.

[56] S.-S. Lim, Y.H. Bae, G.T. Jang, B.-D. Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park, S.M. Moon, and C.S. Kim. An Accurate Worst Case Execution Time Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7), July 1995.

[57] S.-S. Lim, S. L. Min, M. Lee, C. Park, H. Shin, and C. S. Kim. An Accurate Instruction Cache Analysis Technique for Real-Time Systems. In *Proc. of the Workshop on Architectures for Real-Time Applications*, April 1994.

[58] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, 1 July 2003.

[59] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO '09 : Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

[60] lp_solve ILP solver, http ://lpsolve.sourceforge.net/.

[61] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99 : Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[62] Mälardalen benchmarks, http ://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[63] Florian Martin Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In *In Proceedings of the 7th International Conference on Compiler Construction, volume 1383 of Lecture Notes in Computer Science*, pages 80–94. Springer-Verlag, 1998.

[64] S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[65] F. Mueller. Timing predictions for multi-level caches. *Proc. acm sigplan workshop on languages, compilers and tools for real-time systems (lct-rts'97)*, pages 29–36, 1997.

[66] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, pages 209–239, May 2000.

[67] F. Mueller and D. Whalley. Fast instruction cache analysis via static cache simulation. *TR 94042, Dept. of CS, Florida State University*, April 1994.

[68] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. PapaBench : A Free Real-Time Benchmark. In Frank Mueller, editor, *International Workshop on Worst-Case Execution Time Analysis (WCET), Dresden, 04/07/2006*, page (on line), http ://drops.dagstuhl.de/, juillet 2006. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[69] Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[70] Kaustubh Patil, Kiran Seth, and Frank Mueller. Compositional static instruction cache simulation. *SIGPLAN Not.*, 39(7) :136–145, 2004.

[71] S. M. Petters, A. Betts, and G. Bernat. A new timing schema for wcet analysis. In *4th International Workshop on Worst-Case Execution Time Analysis (WCET2004).* IRISA, 2004. internal publication n1645.

[72] William Pugh. Counting solutions to presburger formulas : how and why. In *PLDI '94 : Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 121–134, New York, NY, USA, 1994. ACM.

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

[74] Abdur Rakib, Oleg Parshin, Stephan Thesing, and Reinhard Wilhelm. Component-wise instruction-cache behavior prediction. In *Automated Technology for Verification and Analysis*, pages 211–229, 2004.

[75] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2) :175–188, 1999.

[76] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. *11th ieee real time and embedded technology and applications symposium, 2005. rtas 2005*, pages 148–157, 2005.

[77] Jan Reineke and Daniel Grund. Relative competitiveness of cache replacement policies. In *SIGMETRICS '08 : Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 431–432, New York, NY, USA, 2008. ACM.

[78] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2) :99–122, 2007.

[79] C. Rochange and P. Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Transactions on High-Performance Embedded Architecture and Compilation*, 2(3) :109–128, 2007.

[80] Christine Rochange and Pascal Sainrat. Towards Designing WCET-predictable Processors. In *3rd Workshop on Worst-Case Execution Time Analysis , Porto*, pages 87–90. –, 1 July 2003.

[81] Rathijit 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, New York, NY, USA, 2007. ACM.

[82] SNU-RT benchmarks, http ://archi.snu.ac.kr/realtime/benchmark/.

[83] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical report, 1993.

[84] Robert Tarjan. Testing flow graph reducibility. In *STOC '73 : Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107, New York, NY, USA, 1973. ACM.

[85] A. Tesanovic, J. Hansson, D. Nyström, C. Norström, and P. Uhin. Aspect-Level WCET Analyzer : a Tool for Automated WCET Analysis of a Real-Time Software Composed Using Aspect and Components. In *3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003).*, 1 July 2003.

[86] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *RTSS '98 : Proceedings of the IEEE Real-Time Systems Symposium*, page 144. IEEE Computer Society, 1998.

[87] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3) :157–179, 2000.

[88] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *International Conference on Dependable Systems and Networks*, pages 625–632, 22 June 2003.

[89] Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *DATE '05 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 600–605, Washington, DC, USA, 2005. IEEE Computer Society.

[90] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard 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.

[91] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Design, Automation and Test in Europe (DATE'05)*, volume 1, pages 606–611, 04 2005.

[92] Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches.

In *IEEE Real-Time Technology and Applications Symposium*, pages 192–202. IEEE Computer Society, 1997.

[93] Randall T. White, David B. Whalley, and A. Riccardi. Bounding worst-case data cache performance. Technical report, Florida State University, 1997.

[94] Renhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[95] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0 :80–89, 2008.

Table des figures

Liste des tableaux